(19)日本国特許庁（ＪＰ）　(12) **公開特許公報（Ａ）**　(11)特許出願公開番号
特開2000－181724
（P2000－181724A）
(43)公開日　平成12年６月30日(2000.6.30)

(51)Int.Cl.⁷	識別記号	ＦＩ		テーマコード（参考）
Ｇ０６Ｆ　9/45		Ｇ０６Ｆ　9/44	３２２Ａ ３２０Ｃ	

審査請求　未請求　請求項の数30　ＯＬ　外国語出願　（全 60 頁）

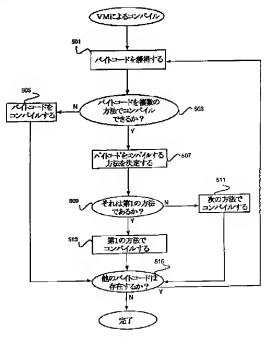
(21)出願番号　特願平11－309657
(22)出願日　平成11年10月29日(1999.10.29)
(31)優先権主張番号　０９／１８３４９９
(32)優先日　平成10年10月30日(1998.10.30)
(33)優先権主張国　米国（ＵＳ）

(71)出願人　591064003
サン・マイクロシステムズ・インコーポレーテッド
ＳＵＮ　ＭＩＣＲＯＳＹＳＴＥＭＳ，ＩＮＣＯＲＰＯＲＡＴＥＤ
アメリカ合衆国　94303　カリフォルニア州・パロ　アルト・サン　アントニオ　ロード・901
(74)代理人　100096817
弁理士　五十嵐　孝雄　（外２名）

最終頁に続く

(54)【発明の名称】　コンパイルする方法をランタイムにおいて選択する方法及び装置

(57)【要約】
【課題】　プログラムをコンパイルする方法をランタイムにおいて決定する装置、方法及びコンピューター・プログラム製品を開示する。
【解決手段】　複数の方法でコンパイルできるプログラムに関連するバイトコード命令を取り出し、特定の方法（一般的には、デフォルトの方法）でコンパイルする。ランタイムにおいて、バーチャル・マシンはバイトコード命令をコンパイルする別の方法がより望ましいか否かを決定し、望ましい場合、バイトコードをこの別の方法でリコンパイルする。幾つかの実施例では、リコンパイルするバイトコード命令を含むプログラムの一部は、リコンパイルする他の命令と一緒にキューへ加えられる。バーチャル・マシンはプログラムの実行時に発生したプログラムの変化する要件を調べる。この場合、これらの要件は、プログラムをコンパイルできる複数の方法のそれぞれのプロフィール・データに由来している。プロフィール・データに基づいて、プログラム内のバイトコード命令を更に好ましい方法でリコンパイルできる。



1

【特許請求の範囲】

【請求項１】　コンピュータ・プログラムに関連するバイトコード命令をコンパイルする方法をランタイムにおいて決定する方法であって、
複数の方法でコンパイルされ得る前記コンピュータ・プログラムに関連するバイトコード命令を取り出す工程と、
前記バイトコード命令を第１の方法でコンパイルする工程と、
前記バイトコード命令をコンパイルする第２の方法が望ましいことを、ランタイムにおいて決定する工程と、
前記バイトコード命令を前記第２の方法でリコンパイルする工程と、を備える方法。

【請求項２】　請求項１に記載の方法において、
リコンパイルされるべき前記バイトコード命令を備えるモジュールを、キューへ加える工程を更に備えることを特徴とする方法。

【請求項３】　請求項１に記載の方法において、
前記バイトコード命令を複数の方法でコンパイルできることを、ランタイムにおいて決定する工程を更に備えることを特徴とする方法。

【請求項４】　請求項１に記載の方法において、
前記コンピュータ・プログラムのダイナミックに変化する効率を調べる工程を更に備えることを特徴とする方法。

【請求項５】　請求項４に記載の方法において、
コンピュータ・プログラムをコンパイルできる前記複数の方法のうちの現在実行中の１つの方法に関する特定のデータを、ランタイムにおいて集める工程を更に備えることを特徴とする方法。

【請求項６】　請求項１に記載の方法において、
前記第２の方法が前記第１の方法と異なる際、前記バイトコード命令を前記第２の方法でリコンパイルする工程を更に備えることを特徴とする方法。

【請求項７】　ネイティブ命令の異なるセットを、ソフトウェア・プログラムから形成する方法であって、
ネイティブ命令の第１のセットを形成するために、前記ソフトウェア・プログラムを第１の方法でコンパイルする工程と、
前記ソフトウェア・プログラムを別の方法でコンパイルすることが望ましいことを、前記ソフトウェア・プログラムのランタイムにおいて決定する工程と、
ネイティブ命令の第２のセットを形成し、これによって、前記ネイティブ命令の第１のセットを前記ネイティブ命令の第２のセットと置換するために、前記ソフトウェア・プログラムを前記別の方法でリコンパイルする工程と、を備える方法。

【請求項８】　請求項７に記載の方法において、
前記ソフトウェア・プログラムをリコンパイルするのにどの方法が有利かを決定するために、ダイナミックに形

2

成された特定のデータを調べる工程を更に備えることを特徴とする方法。

【請求項９】　請求項８に記載の方法において、
前記ダイナミックに形成された特定のデータは、特定のコンパイル方法を実行した回数を格納するカウンタを含むことを特徴とする方法。

【請求項１０】　請求項７に記載の方法において、
前記ソフトウェア・プログラムのダイナミックに変化する要件をより効果的に処理するために、前記ネイティブ命令の第１のセットが前記ネイティブ命令の第２のセットによって置換されるべきか否かを決定する工程を更に備えることを特徴とする方法。

【請求項１１】　請求項７に記載の方法において、
前記ソフトウェア・プログラムのランタイムにおいて、
前記ソフトウェア・プログラム内のどの特定の命令がコンパイルされるべきかを決定し、前記特定の命令をマークする工程を更に備えることを特徴とする方法。

【請求項１２】　プログラム内の浮動小数点オペレーションを実行する方法であって、
浮動小数点オペレーションが浮動小数点アンダフローを形成し得るか否かを決定する工程と、
特定の浮動小数点オペレーションがアンダフローを何回引き起こしたかを決定するために、特定のインジケータをチェックする工程と、
前記特定のインジケータが所定の基準を満たした時、前記特定の浮動小数点オペレーションを第１の方法を用いてコンパイルし、それ以外の時は、前記特定の浮動小数点オペレーションを第２の方法を用いてコンパイルする工程と、を備える方法。

【請求項１３】　請求項１２に記載の方法において、
前記特定の浮動小数点オペレーションをコンパイルする第１及び第２の方法に関するデータを、ランタイムにおいてダイナミックに形成し、格納する工程を更に備えることを特徴とする方法。

【請求項１４】　請求項１２に記載の方法において、
前記第１の方法はトラップ・ルーチンであり、前記第２の方法は明示的チェックであり、前記明示的チェックはインライン・コードを前記プログラムへ挿入することによってインプリメントされることを特徴とする方法。

【請求項１５】　請求項１４に記載の方法において、
前記特定の浮動小数点オペレーションが明示的チェックを用いてコンパイルされた時点から所定時間が経過したか否かをチェックする工程と、
前記所定時間が経過している場合、リコンパイルされるべき前記特定の浮動小数点オペレーションを含むモジュールをマークする工程と、を更に備えることを特徴とする方法。

【請求項１６】　請求項１４に記載の方法において、
前記トラップ・ルーチンを開始する工程を更に備えることを特徴とする方法。

【請求項１７】　請求項１４に記載の方法において、前記特定の浮動小数点オペレーションが明示的チェックを用いてコンパイルされる場合、タイマをセットする工程を更に備えることを特徴とする方法。

【請求項１８】　請求項１４に記載の方法において、前記特定のインジケータは、前記浮動小数点アンダフローがトラップ・ルーチンを用いて処理される度に、インクリメントされるカウンタであることを特徴とする方法。

【請求項１９】　請求項１４に記載の方法において、どの浮動小数点オペレーションが前記トラップ・ルーチンの実行を引き起こしたかを決定する工程と、
カウンタをインクリメントする工程と、
前記カウンタが所定値を超えている場合、リコンパイルされるべき前記特定の浮動小数点オペレーションを含むモジュールをマークする工程と、
を備えることを特徴とする方法。

【請求項２０】　請求項１９に記載の方法において、前記モジュールがリコンパイルすべくマークされている場合、前記モジュールをリコンパイル・キューへ加える工程を更に含む請求項１９に記載の方法。

【請求項２１】　請求項１９に記載の方法において、前記方法がリコンパイルされるべくマークされている場合、前記方法に関連するカウンタをリセットする工程を更に備えることを特徴とする方法。

【請求項２２】　浮動小数点アンダフローを検出する命令を形成する方法であって、
プログラム内のオペレーションが浮動小数点アンダフローを形成し得るか否かを決定する工程と、
アンダフローを引き起こすための特定の浮動小数点オペレーションの傾向を測定するために、アンダフロー・インジケータを調べる工程と、
前記アンダフローを引き起こすための前記特定の浮動小数点オペレーションの傾向に従って、前記特定の浮動小数点オペレーションをトラップ・ルーチンを用いてコンパイルする工程と、
前記アンダフローを引き起こすための前記特定の浮動小数点オペレーションの傾向に従って、前記浮動小数点オペレーションを明示的インライン・チェックを用いてリコンパイルする工程と、
を備える方法。

【請求項２３】　請求項２２に記載の方法において、前記トラップ・ルーチン及び前記明示的インライン・チェックに関連するデータを、ランタイムにおいてダイナミックに形成し、格納する工程を更に備えることを特徴とする方法。

【請求項２４】　請求項２２に記載の方法において、前記特定の浮動小数点オペレーションは、モジュール内に含まれていると共に、
リコンパイルされるべき前記浮動小数点オペレーションを含む前記モジュールを、リコンパイル・キューへ加える工程を更に備えることを特徴とする方法。

【請求項２５】　コンピュータ・プログラムに関連するバイトコード命令をコンパイルする方法をランタイムにおいて決定するコンピュータ・プログラム製品であって、
複数の方法でコンパイルされ得る前記コンピュータ・プログラムに関連するバイトコード命令を取り出すコンピュータ・コードと、
前記バイトコード命令を第１の方法でコンパイルするコンピュータ・コードと、
前記バイトコード命令をコンパイルする第２の方法が望ましいことを、ランタイムにおいて決定するコンピュータ・コードと、
前記バイトコード命令を前記第２の方法でリコンパイルするコンピュータ・コードと、
前記コンピュータ・コードを格納するコンピュータ読取り可能媒体と、
を備えるコンピュータ・プログラム製品。

【請求項２６】　プログラム内の浮動小数点オペレーションを実行するコンピュータ・プログラム製品であって、
浮動小数点オペレーションが浮動小数点アンダフローを形成し得るか否かを決定するコンピュータ・コードと、
特定の浮動小数点オペレーションがアンダフローを何回引き起こしたかを決定するために、特定のインジケータをチェックするコンピュータ・コードと、
前記特定のインジケータが所定の基準を満たした時、前記特定の浮動小数点オペレーションを第１の方法を用いてコンパイルし、それ以外の時は、前記浮動小数点オペレーションを第２の方法を用いてコンパイルするコンピュータ・コードと、
前記コンピュータ・コードを格納するコンピュータ読取り可能媒体と、
を備えるコンピュータ・プログラム製品。

【請求項２７】　コンピュータ・プログラムに関連するバイトコード命令をコンパイルする方法をランタイムにおいて決定するシステムであって、
複数の方法でコンパイルされ得る前記ソフトウェア・プログラムに関連するバイトコード命令を取り出すバイトコード・リトリーバと、
前記バイトコード命令を第１の方法及び第２の方法のうちの一方でコンパイリングするコンパイリング・モジュールと、
前記命令をコンパイルする前記第２の方法が望ましいことを、ランタイムにおいて決定するオルタネイティブ・コンパイリング・ディテクタと、
を備えるシステム。

【請求項２８】　請求項２７に記載のシステムにおいて、

１つ以上のモジュールを保持するモジュール・キューを更に備え、
各モジュールはリコンパイルされるべきバイトコード命令を含むことを特徴とするシステム。
【請求項２９】　請求項２７に記載のシステムにおいて、
前記コンピュータ・プログラムのダイナミックに変化する効率を調べるための効率アナライザ・イグザミナを更に備えることを特徴とするシステム。
【請求項３０】　請求項２７に記載のシステムにおいて、
前記コンピュータ・プログラムがコンパイルされ得る前記複数の方法のうちの現在実行中の１つの方法に関する特定のデータを、ランタイムにおいて集めるランタイム・データ・コレクタを更に備えることを特徴とするシステム。
【請求項３１】　プログラム内の浮動小数点命令を実行するシステムであって、
浮動小数点命令が浮動小数点アンダフローを形成し得るか否かを決定する命令エバリュエータと、
特定の浮動小数点オペレーションが浮動小数点アンダフローを引き起こした回数を保持する浮動小数点アンダフロー・インジケータと、
前記特定のインジケータが所定の基準を満たした時、前記特定の浮動小数点オペレーションを第１の方法を用いてコンパイルし、それ以外の時は、前記特定の浮動小数点オペレーションを第２の方法を用いてコンパイルするコンパイラを更に備え、
前記特定の浮動小数点オペレーションがアンダフローを何回引き起こしたかを決定するために、前記浮動小数点アンダフロー・インジケータがチェックされることを特徴とするシステム。
【請求項３２】　請求項３１に記載のシステムにおいて、
前記浮動小数点オペレーションをコンパイルする第１及び第２の方法に関するデータを、ランタイムにおいてダイナミックに形成し、格納するデータ・ジェネレータを更に備えることを特徴とするシステム。
【請求項３３】　命令を含むプログラムをコンパイルする方法をランタイムにおいて決定するシステムであって、
１つ以上のプロセッサと、
前記１つ以上のプロセッサによって実行されるプログラムを格納するコンピュータ読取り可能媒体と、
を備え、
前記コンピュータ読取り可能媒体は、
複数の方法でコンパイルされ得る前記プログラム内の命令を取り出すコンピュータ・コードと、
前記命令を第１の方法でコンパイルするコンピュータ・コードと、
前記命令をコンパイルする第２の方法が望ましいことを、ランタイムにおいて決定するコンピュータ・コードと、
前記命令を前記第２の方法でリコンパイルするコンピュータ・コードと、
を備えることを特徴とするシステム。
【発明の詳細な説明】
【０００１】
【発明の属する技術分野】一般的に、本発明はコンピュータ・ソフトウェア及びソフトウェア・ポータビリティの分野に関する。特に、本発明はプログラムをプラットフォーム固有の要件に基づいてコンパイルする方法に関する。
【０００２】
【従来の技術】ジャバ（Ｊａｖａ（商標名））バーチャル・マシン（ＪＶＭ）は、様々なコンピュータ・アーキテクチャ上でインプリメント可能であり、異なるマイクロプロセッサからそれぞれ生じる異なる仕様及び規格に適応可能である。この種の規格の１つの例としては、いくつかのマイクロプロセッサが浮動小数点数を扱う際に利用する拡張精度浮動小数点フォーマットが挙げられる。この種のマイクロプロセッサの１つとしては、拡張精度（８０ビット）浮動小数点演算を使用するインテル社のＩＡ－３２マイクロプロセッサ・アーキテクチャが挙げられる。一般的に、他のプロセッサは単（３２ビット）精度浮動小数点演算または倍（６４ビット）精度浮動小数点演算を使用する。
【０００３】拡張精度フォーマットで算出された数値を単精度フォーマットまたは倍精度フォーマットへ変換する際、問題が発生する。浮動小数点オペレーションが結果をＩＥＥＥ７５４の指定するレンジ及び精度で形成すべきことは、ジャバ（商標名）言語によって指定されている。このＩＥＥＥ７５４の内容は、この開示をもって本明細書中に開示したものとする。その一方、カリフォルニア州サンタクララに所在するインテル社が製造するインテルＩＡ－３２プロセッサは結果をより広いレンジ及びより高い精度で形成する。これによって、前記の問題が発生する。これらのより広い結果はＩＥＥＥ７５４単精度フォーマット及び倍精度フォーマットへ正確に丸める必要がある。ＩＡ－３２マイクロプロセッサの場合、この種の丸めを実施する少なくとも２つの方法があり、これらの方法はそれぞれ異なるコスト（コード・サイズ及び実行速度）を伴う。スタティック・コンパイラ（またはワンタイム・ダイナミック・コンパイラ）は、１つのインプリメンテーションを選択する必要があり、この選択は全ての状況下で最適な選択とはならない。前記の問題を図１に示す。
【０００４】図１は倍精度浮動小数点の一般的なフォーマットと、拡張精度浮動小数点のフォーマットとを示すブロック図である。フォーマット１０２は、インテルＩ

Ａ－３２アーキテクチャの倍精度浮動小数点フォーマットとは対照的なインテルＩＡ－３２アーキテクチャの拡張精度浮動小数点数フォーマットを示す。符号ビット１０４は、この浮動小数点数が正及び負のいずれであるかを示す。浮動小数点数の指数の値を表すための指数値を示すビット１０６が、これに続いて配置されている。

【０００５】ビット１０８は仮数を保持するためのビットを含む。仮数部は浮動小数点数の整数部を表すビットを最高で６４ビット保持できる。従って、８０（１＋１５＋６４）ビットが、拡張精度浮動小数点フォーマット内に存在する。一般的に、浮動小数点オペレーションは浮動小数点ユニットによって処理される。仮数及び指数を操作することによって、このユニットは浮動小数点数を必要とする複雑なオペレーションを効率的に実施できる。当該技術分野でよく知られているように、浮動小数点数を整数及び指数で表すことにより、浮動小数点数の演算は遙かに容易になる。

【０００６】更に、図１はＩＥＥＥ７５４に開示されている倍精度浮動小数点フォーマット１１２を示す。このフォーマットは拡張精度フォーマットとレイアウトの点で類似しているが、指数フィールド及び仮数フィールド内のビット数の点で異なる。前記のように、インテルＩＡ－３２プロセッサは結果を拡張精度フォーマットで形成する。前記の問題はジェームズ・ゴスリン、ビル・ジョイ及びガイ・スチールによる“ジャバ（商標名）言語詳説”（ＩＳＢＮ０－２０１－６３４５１－１）に由来しており、この文献の内容はこの開示をもって本明細書に開示したものとする。この詳説では、結果をＩＥＥＥ７５４単精度フォーマットまたは倍精度フォーマットで形成する必要がある。フォーマット１１２の説明へ戻り、符号ビット１０４は、倍精度フォーマットまたは単精度フォーマットと対照をなす拡張精度フォーマットにおける符号ビットと同じである。指数ビット１１４はビット１０６と同じ機能を有するが、拡張精度フォーマットにおける１５ビットとは対照的に１１ビットを保持する。仮数ビット１１６は拡張精度フォーマットにおける６４ビットとは対照的に５２ビットを保持する。従って、倍精度浮動小数点フォーマットは６４ビットを保持可能である。仮数長の違いを、図１の破線領域１１８で示す（指数長における４ビットの違いは図中にこれと同様に示されていない）。

【０００７】拡張精度の結果が例えばＩＡ－３２プロセッサから与えられた際、単精度フォーマットまたは倍精度フォーマットの結果を必要とするジャバ言語では、問題が発生する。拡張指数が単精度または倍精度のレンジの外側に位置する場合、オーバフローまたはアンダフローが発生し得る。ＩＡ－３２では、オーバフローはハードウェアによって処理されるが、本発明の方法で解決に努めることができる。指数を減らすべく、仮数が（右側へ）シフトするので、アンダフローは処理が更に困難である。しかしながら、このシフトによって仮数のビットが失われ、これによって、結果の精度が失われる。正しく、かつ、精度が更に低い仮数を演算するためには、幾つかの命令が必要であり、一般的に、オペレーションは必要な時に呼び出されるように、別個のサブルーチンに置かれる。前記の問題は結果を正しく丸めることではなく、むしろ、訂正が生ずべきことを検出することにある。

【０００８】図２（ａ）及び図２（ｂ）は浮動小数点アンダフローを検出する２つの方法を示す。図２（ａ）に示す１つの方法では、問題を訂正するトラップ・ルーチン２０６を呼び出すために、プログラム・コード２０２内のトラップ・ハンドラ２０４を使用して、トラップを実施することによって、プログラム・コード２０２は、その問題を検出する。図２（ｂ）に示す別の方法では、プログラム・コード２０８はコード２１０を含んでおり、そのコード２１０は、乗算オペレーション及び除算オペレーションなど、問題を潜在的に引き起こし得る方法で、浮動小数点が使用される度、その問題を検出する。

【０００９】トラップ・ハンドラ２０６を利用することによって、問題の解決に努めている間、全てのオペレーションは中止される。トラップを呼び出す際、トラップ・ルーチンの実行前に、トラップを引き起こした命令のロケーションを含むマシンの状態を格納する。しかし、トラップを呼び出さない場合、オペレーション毎のオーバーヘッドは存在しない。そして、トラップ・ハンドラをセットアップするための、ワンタイム・オーバーヘッドがスレッド毎に存在するのみである。そして、このセットアップされたトラップ・ハンドラは全ての浮動小数点オペレーションを監視する。その一方、プログラム・コード２１０は、浮動小数点アンダフローの問題を処理するために、コードをプログラムへ挿入する技術を示す。この方法では、正しく丸められた結果を形成するために、必要に応じて、問題をチェックし、サブルーチンを呼び出す目的で、インライン・コードがアンダフローの問題を引き起こし得る各オペレーションの後に続けられる。この方法は、発生しない各アンダフローに対する多くの不必要なプロセッサ・オペレーションを必要とする。しかし、その問題が検出された際、この問題は、全てのオペレーションを一時停止させることと、トラップを処理するために、プロセス・コンテキストまたはスレッド・コンテキストを保存することを要することなく、解決される。前記のように、浮動小数点アンダフローは、任意のプラットフォーム上における選択可能な解決策を有する問題の１つの例である。他の問題が発生し得る。この場合、ジャバ・バーチャル・マシンは特定の問題を解決するいくつかのインプリメンテーションを利用可能であり、各インプリメンテーションはいくらかのケースでより効率的である。

【００１０】
【発明が解決しようとする課題】従って、プラットフォーム固有のバリエーションに起因して生じる問題の解決に使用するために、インプリメンテーションをインテリジェントに、ダイナミックに選択することが望ましい。浮動小数点アンダフローの問題を単なる例として使用した場合、例えば、必要なインライン・コードの量を減少し、さらに問題を訂正するサブルーチンをディスパッチするための、トラップ・ハンドラの使用のオーバーヘッドを防止するとともに、浮動小数点アンダフローを検出し、訂正することが望ましい。ダイナミック・ランタイム・コンパイラが１つのインプリメンテーションを選択し、その効率を監視し、要求されれば、そのインプリメンテーションを変更できることが望ましい。

【００１１】
【課題を解決するための手段およびその作用・効果】バーチャル・マシンによって、プログラム実行中に、プログラムに関連するバイトコードをコンパイルする方法を決定する本発明に従う方法、装置及びコンピュター・プログラム製品を開示する。本発明の１つの態様では、複数の方法でランタイムにおいてコンパイルできるプログラム内の命令を取り出し、特定の方法（一般的には、デフォルトの方法）でコンパイルする。次いで、バーチャル・マシンは命令をコンパイルする別の方法がより望ましいことをランタイムにおいて決定し、バイトコード命令はこの別の方法でリコンパイルされる。

【００１２】１つの形態では、リコンパイルされるバイトコード命令を含む実行可能なコードは、リコンパイルされる他の命令と一緒にキューへ加えられる。バーチャル・マシンはプログラムの実行時に発生したプログラムの全ての変化する要件を調べる。この場合、これらの要件は、プログラムをコンパイルできる複数の方法のそれぞれのプロフィール・データに由来している。別の形態では、命令をコンパイルする第１の方法、即ち、デフォルトの方法とは異なる方法で、前記の特定のバイトコード命令をバーチャル・マシンによってリコンパイルする。

【００１３】本発明の別の態様では、単一のプログラムから、実行可能な命令の異なるセットを形成する方法を提供する。バイトコード命令の１つのセットを形成するために、プログラムを特定の方法（デフォルトの方法など）でランタイムにおいてコンパイルする。次いで、バーチャル・マシンはプログラムを別の方法でコンパイルすることが望ましいか否かをランタイムにおいて決定する。そして、バーチャル・マシンは、そのようにして、ネイティブ命令の別のセットを形成し、この別のセットは第１のセットと置換される。

【００１４】１つの形態では、プログラムをリコンパイルする方法を決定するために、バーチャル・マシンは、プログラムを実行できる複数の方法のそれぞれのダイナミックに形成されたプロフィール・データを調べる。プロフィール・データは、プログラムを特定の方法で実行した回数を格納するカウンタを含む。プログラムのダイナミックに変化する要件をより効率的に処理するために、バーチャル・マシンは、ネイティブ命令の特定のセットをネイティブ命令の別のセットと置換すべきか否かを決定する。

【００１５】本発明の別の態様では、プログラム内の浮動小数点命令を実行するシステムを開示する。システムは、特定の命令が浮動小数点アンダフローを形成し得るか否かを決定する。次いで、浮動小数点オペレーションがアンダフローを引き起こした回数を決定するために、システムはインジケータをチェックする。インジケータが所定値未満である場合、バーチャル・マシンは浮動小数点オペレーションを１つの方法でランタイムにおいてコンパイルする。インジケータが所定値を越えている場合、バーチャル・マシンは浮動小数点オペレーションを別の方法でランタイムにおいてコンパイルする。

【００１６】本発明の更に別の態様では、トラップ・ルーチンまたは明示的チェックを使用して浮動小数点アンダフローを検出するための命令を形成する方法を開示する。プログラム内のオペレーションが浮動小数点アンダフローを形成し得るか否かを決定する。次いで、特定の浮動小数点オペレーションがアンダフローを引き起こした回数を決定するために、カウンタをチェックする。カウンタが所定値未満である場合、オペレーションをトラップ・ルーチンを用いてランタイムにおいてコンパイルする。カウンタが所定値を越えている場合、オペレーションを明示的インライン・チェックを用いてリコンパイルする。トラップ・ルーチンに関連するデータ及び明示的インライン・チェックに関連するデータを、ランタイムにおいてダイナミックに形成し、格納する。

【００１７】本発明は添付図面に基づく以下の説明によって更によく理解できる。

【００１８】
【発明の実施の形態】本発明の特定の実施例を詳述する。この実施例は添付図面に示されている。本発明を特定の実施例に関連して詳述するが、これは本発明を１つの実施例に限定することを意図するものではない。逆に、添付の請求の範囲によって定義される本発明の精神及び範囲内に含まれる別例、変更例及び等価なものを包含することを意図している。

【００１９】本発明は、特定の種類のオペレーションを任意のアーキテクチャ上で実施する選択可能な複数のインプリメンテーションの選択に取り組む。一般的に、従来の方法はコードをコンパイル・タイムにおいてスタティックに一度形成するか、またはランタイムにおいてダイナミックに一度形成する。ここで開示され、クレームされた発明は、バーチャル・マシンが、ランタイム・コンパイラによって形成するコード・セグメントを、複数

の可能なコード・セグメントの中からランタイム・パフォーマンス・データに基づいてランタイムで選択することを可能にする。これによって、ダイナミック・コンパイラは１つのインプリメンテーションを選択でき、要求されれば、その効率を監視し、インプリメンテーションを変更する。

【００２０】前記のように、浮動小数点アンダフローを検出して修正する２つの一般的な方法が存在する。このうちの一方の方法は、高速で短いコードであって、正常なプログラムの実行の中断をトラップの処理中に必要とするコードを使用することを含むトラップ法を使用する。他方の方法は、アンダフローを各浮動小数点オペレーション後に検出するために、コードをプログラム内に挿入することを含む。これは、コードを毎回実行することを必要とするが、プログラムを順番に続行することを可能にする。

【００２１】拡張浮動小数点フォーマットの結果を単精度浮動小数点フォーマットまたは倍精度浮動小数点フォーマットで格納する必要が生じた際、アンダフローの問題が発生する。これは、例えば、結果をインテル・アーキテクチャ・コンピュータ上で算出し、次いで、この結果を単精度フォーマットまたは倍精度フォーマットで格納する際、発生し得る。より具体的には、この問題は、結果の指数が宛先において表示可能な最小の指数（ＩＥＥＥ７５４単精度フォーマットまたは倍精度フォーマット）より小さく、仮数が正確な結果を丸めた結果である際など、非常に小さな数を使用した演算を実施する際に、発生する。最も近い表現を宛先内に格納するために、仮数が不正確であること（その結果、丸められたこと）と、仮数を丸めた方法及び理由と、を知る必要がある。この検出及び訂正は、例えば数値がゼロに近づく速度を測定すべく、非常に小さな数の正確さを維持するために、重要である。

【００２２】図３は本発明の一実施例に従う、ジャバ・ソース・コードからネイティブ命令を形成することに関連した入力／出力及び実行ソフトウェア／システムを示すブロック図である。他の実施例では、本発明を、別の言語のためのバーチャル・マシンを用いて実施するか、またはジャバ・クラス・ファイル以外のクラス・ファイルを用いて実施できる。図の左側から始めると、第１入力は、カリフォルニア州マウンテンビューに所在するサン・マイクロシステムズによって開発されたジャバ（商標名）プログラム言語で書かれたジャバ・ソース・コード３０１である。ジャバ・ソース・コード３０１をバイトコード・コンパイラ３０３へ入力する。本質的に、バイトコード・コンパイラ３０３はソース・コード３０１をバイトコードへコンパイルするプログラムである。バイトコードは１つ以上のジャバ・クラス・ファイル３０５に含まれる。ジャバ・クラス・ファイル３０５はジャバ・バーチャル・マシン（ＪＶＭ）を有する任意のコンピュータ上で実行できるので、ポータブルといえる。バーチャル・マシンの複数のコンポーネントを図４により詳細に示す。ジャバ・クラス・ファイル３０５はＪＶＭ３０７へ入力される。ＪＶＭ３０７は任意のコンピュータ上に存在可能である。従って、ＪＶＭ３０７は、バイトコード・コンパイラ３０３を有する同一のコンピュータ上に存在する必要はない。ＪＶＭ３０７はインタプリタまたはコンパイラなどの幾つかの役割のうちの１つとしての動作が可能である。それがコンパイラとして動作する場合、それは“ジャスト・イン・タイム（ＪＩＴ）”コンパイラまたはアダプティブ・コンパイラとしてさらに動作し得る。インタプリタとして動作する際、ＪＶＭ３０７はジャバ・クラス・ファイル３０５に含まれる各バイトコード命令をインタプリトする。

【００２３】図４は後で記述する図１１のコンピュータ・システム１０００によってサポートできるＪＶＭ３０７などのバーチャル・マシン３１１を示す図である。前記のように、コンピュータ・プログラム（例：ジャバ（商標名）プログラム言語で書かれたプログラム）をソースからバイトコードへ翻訳する際、ソース・コード３０１はコンパイルタイム環境３０３内のバイトコード・コンパイラ３０３へ提供される。バイトコード・コンパイラ３０９は、ソース・コード３０１をバイトコード３０５へ翻訳する。一般的に、ソフトウェア開発者がソース・コード３０１を形成した時点において、ソース・コード３０１はバイトコード３０５へ翻訳される。

【００２４】一般的に、バイトコード３０５はネットワーク（例：図１１のネットワーク・インターフェース１０２４）を通じて複製、ダウンロード若しくは配布されるか、または図１１の一次ストレージ１００４などのストレージ・デバイス上へ格納され得る。本実施例では、バイトコード３０３はプラットフォームから独立している。即ち、バイトコード３０３は、適切なバーチャル・マシン３１１を実行している実質的に全てのコンピュータ・システム上で実行可能である。バイトコードをコンパイルすることによって形成されたネイティブ命令は、後からＪＶＭで使用するために保持できる。この結果、インタプリトされたコードより優れた速度の効果をネイティブ・コードへ提供するために、翻訳のコストは複数の実行を通じて償却される。例えば、ジャバ（商標名）環境では、バイトコード３０５はＪＶＭを実行している コンピュータ・システム上で実行可能である。

【００２５】バイトコード３０５はバーチャル・マシン３１１を含むランタイム環境３１３へ提供される。一般的に、ランタイム環境３１３は図１１のＣＰＵ１００２などのプロセッサを使用して実行できる。バーチャル・マシン３１１はコンパイラ３１５、インタプリタ３１７及びランタイム・システム３１９を含む。一般的に、バイトコード３０５はコンパイラ３１５またはインタプリタ３１７へ提供可能である。

【００２６】バイトコード３０５をコンパイラ３１５へ提供した際、バイトコード３０５に含まれるメソッドはネイティブ・マシン命令（図示せず）へコンパイルされる。その一方、バイトコード３０５をインタプリタ３１７へ提供した際、バイトコード３０５は１バイトコードずつインタプリタ３１７内へ読み込まれる。そして、各バイトコードがインタプリタ３１７内へ読み込まれることにより、インタプリタ３１７は各バイトコードによって定められたオペレーションを実施する。一般的に、インタプリタ３１７は実質的に連続してバイトコード３０５を処理し、バイトコード３０５に関連するオペレーションを実施する。

【００２７】オペレーティング・システム３２１がメソッドを呼び出す際、このメソッドをインタプリトされたメソッドとして呼び出すことを決定した場合、ランタイム・システム３１９はメソッドをインタプリタ３１７から獲得できる。その一方、メソッドをコンパイルされたメソッドとして呼び出すことを決定した場合、ランタイム・システム３１９はコンパイラ３１５を起動する。次いで、コンパイラ３１５はネイティブ・マシン命令をバイトコード３０５から形成し、マシン言語命令を実行する。一般的に、バーチャル・マシン３１１を終了する際、マシン言語命令は廃棄される。バーチャル・マシン、より詳細には、ジャバ（商標名）バーチャル・マシンのオペレーションはティム・リンドホルム及びフランク・イェリンによる“ジャバ（商標名）バーチャル・マシン詳説”（ＩＳＢＮ０－２０１－６３４５２－Ｘ）と称される文献に更に詳細に開示されており、この文献の内容はこの開示をもって本明細書中に開示したものとする。

【００２８】前記のように、ジャバ・プログラム内の命令は時には１より多い方法でコンパイルできる。先の例の続きを説明する。乗算（ＦＭＵＬ）オペレーションまたは除算（ＦＤＩＶ）オペレーションなどのアンダフローを潜在的に引き起こし得る浮動小数点オペレーションは、少なくとも２つの方法（明示的チェックを伴う方法またはトラップを伴う方法）でコンパイル可能である。図５はジャバ・プログラムからネイティブ命令の異なるバージョンがどのように形成され得るかを示す流れ図である。バイトコード・コンパイラによってジャバ・クラス・ファイル内にコンパイルされた後、ブロック４０３（図３のシステム３０７）において、ジャバ・プログラム４０１（図３及び図４の入力３０１）は、ＪＶＭによってバイトコードからネイティブ・マシン命令へランタイムでコンパイルされる。ＪＶＭは例示を目的として使用しているだけである。当業者に知られているように、バーチャル・マシンは任意の入力表現からネイティブ命令セットへの一般的な翻訳に用いる。この場合、インプリメンテーションの選択が存在する。ＪＶＭによるジャバ・クラス・ファイルのコンパイレーションの方法を図６に基づいて以下に記述する。

【００２９】前記のように、ＪＶＭは２つ役割、即ち、クラス・ファイルに含まれるジャバ・バイトコードをインタプリトすることと、クラス・ファイルをコンパイルし、これによって、ＪＶＭを有する同一コンピュータ上で実行されるネイティブ命令セット（即ち、これらはポータブルでない）を形成することのうち、いずれか一方の役割を担うことが可能である。従って、コンパイラとして動作するＪＶＭの場合、ブロック４０５に示すように、ＪＶＭがバイトコードをどのようにコンパイルするかに依存して、様々なネイティブ命令セットが同一のジャバ・プログラムから形成され得る。浮動小数点オペレーションを例として使用した場合、ネイティブ命令４０７はその全てのＦＭＵＬ及びＦＤＩＶにおける明示的チェック（即ち、インライン）を含むことが可能な一方、ネイティブ命令４０９はこれら同じ浮動小数点オペレーションのためのトラップのみを含むことが可能であり、また、ネイティブ命令４１１はこれら両方の組み合わせを含むことが可能である。

【００３０】どのコンパイレーション・ルートを採用するか（即ち、ジャバ・ソース・コードをコンパイルすることと、ジャバ・ソース・コードをインタプリトすることと、どのようにもしくはいつジャバ・ソース・コードを実行するかについて他のオペレーションを実施することのうちのどれか）をランタイムにおいて決定するＪＶＭを、図５が示していないことは注目に値する。これに代えて、ＪＶＭの採用したコンパイレーション・ルートが、そのコードをランタイムにおいてコンパイルすることである場合、それを異なる“方法”で実行し、これによって、ネイティブ命令の異なるセットを形成することを、図５は示している。このプロセスを図６に関連して詳述する。

【００３１】図６は本発明の一実施例に従うジャバ・バイトコードをネイティブ・マシン命令へコンパイルするジャバ・バーチャル・マシンのプロセスを示すフローチャートである。ステップ５０１では、ＪＶＭは１つ以上のバイトコード命令をジャバ・クラス・ファイルから取り出す。ステップ５０３では、ＪＶＭは特定の命令を複数の方法でコンパイルできるか否かを決定する。複数の方法でコンパイルできるバイトコード命令の特定の例は図８に基づいて後で詳述する。ＪＶＭが、ＩＡＤＤオペレーションまたはＬＳＵＢオペレーションのように、１つの方法でしか命令をコンパイルできないことを決定した場合、ＪＶＭはバイトコードをステップ５０５でコンパイルする。以前に取り出したバイトコードをコンパイルした後、ＪＶＭは残されたバイトコードが存在するか否かをステップ５１５で決定する。

【００３２】バイトコードをコンパイルする複数の方法が存在することを、ＪＶＭが決定した場合、ＪＶＭはどの方法でバイトコードをコンパイルするかをステップ５

０７で決定すべく処理を続行する。記述した実施例では、図７に詳細を示すように、この決定を行うために、ＪＶＭはメカニズムを使用する。このメカニズムは、バイトコード命令をコンパイルできる異なる方法のそれぞれのダイナミックに形成されたプロフィール情報を使用することを含む。ステップ５０９では、ＪＶＭは、バイトコードをデフォルトの方法を用いてコンパイルするか否かを決定する。そのデフォルトの方法は、一般的に、ランタイム・コンパイラのライタが、その時点で利用可能なオプションを検討した後、最も効率的または論理的な方法であると確信する方法である。

【００３３】ステップ５０９で、ＪＶＭがバイトコードを第１の方法でコンパイルすることを決定した場合、ステップ５１３で、ＪＶＭはこれを実施して、図５のネイティブ命令セットＡなどの第１ネイティブ命令セットを形成する。次いで、ＪＶＭは、他のバイトコードがクラス・ファイル内に存在するか否かをステップ５１５で決定する。存在する場合、ＪＶＭは、次のバイトコードを取り出すためにステップ５０１へ戻る。バイトコードが１つも存在しない場合、プロセスは完了する。ステップ５０９で、ＪＶＭが、バイトコードをコンパイルする方法が第１の方法、即ち、デフォルトの方法でないことを決定した場合、ＪＶＭはバイトコード命令を別のコンパイレーション技術を使用してステップ５１１でコンパイルする。次いで、ＪＶＭはステップ５１３からの場合と同様に処理を続行し、コンパイルする残りのバイトコードが存在するか否かをステップ５１５において決定する。簡単にするために、２つの異なる方法のみを図６に示すが、本発明はバイトコードをコンパイルする３つ以上の方法へ効果的に適用できる。

【００３４】図７は本発明の一実施例に従うダイナミックに形成されたプロフィール・データを含むネイティブ・マシン命令をどのように形成するかを示すブロック図である。ＪＶＭによってバイトコード命令をコンパイルし得る異なる方法のそれぞれに関する情報を含む点を除けば、図７は図５に類似している。その情報は、図６のステップ５０３に示すように、バイトコードをネイティブ命令へコンパイルする複数の方法が存在することが決定されると、形成される。ジャバ・プログラム６０１が図７のトップに位置している。バイトコード・コンパイラによってバイトコードへコンパイルされた後、ジャバ・プログラム６０１はジャバ・バーチャル・マシン６０３へ入力される。次いで、ＪＶＭは、バイトコードをネイティブ命令へコンパイルし得る異なる方法に基づいて、幾つかの異なるネイティブ命令セット６０５を出力可能である。ネイティブ命令セット６０５は、ランタイムにおいてダイナミックに集められたデータ６０７を格納するデータ・スペースを更に含むことができる。この情報は、カウンタ、タイミング・データ、及びバイトコードをコンパイルする特定の方法の効率に関する他の情報のようなプロフィール情報を含んでよい。

【００３５】ダイナミックに集められたデータは、ネイティブ命令と一緒に格納可能であり、そして、バイトコードをＪＶＭによってコンパイルする間に更新できる。１つの実施例では、図６のステップ５０７で最初に説明したように、ＪＶＭは、バイトコードをどの方法でコンパイルするかを決定するために、この情報を調べる。ダイナミック・プロフィール・データは、例えば、コンパイルの特定の方法が効率的であり続けるか否か、バイトコードがその方法で何回実行されたか、または特定の時間を経過したか否かを決定するために、ＪＶＭによって使用され得る。ＪＶＭは、現在の命令がバイトコードの最も効率的なインプリメンテーションであるか否かを決定するために、コンパイル中におけるデータへのクエリーが可能である。効率的に実行されていないことがＪＶＭによって確認された任意のバイトコードを、ＪＶＭによってリコンパイルできる。ＪＶＭは、データ６０７へのクエリーによってどのようにバイトコードがコンパイルされるべきかを決定すると、図６のステップ５０９に示すように、ＪＶＭはこれが第１の方法（デフォルトの方法）であるべきか、または他の方法のうちの１つであるべきかを決定することによって、処理を継続し得る。

【００３６】図８は本発明の記述した実施例に従う、浮動小数点オペレーションをコンパイルし、アンダフローが発生した場合、このアンダフローを訂正する方法を決定するジャバ・バーチャル・マシンを示すフローチャートである。前記のように、浮動小数点オペレーションのコンパイルは、プログラムを幾つかの方法でコンパイルする方法を決定する特定の例である。より一般的には、コンパイレーションをヒューリスティックス、即ち、コード（例：テーブルスイッチ命令のコンパイル）の平均的振る舞いに関する仮定によってガイドする任意のアプリケーションは、前記のようにプログラムを幾つかの方法でコンパイルする方法を決定する方法を利用できる。ステップ７０１では、ＪＶＭはバイトコードをジャバ・クラス・ファイルから取り出す。ステップ７０３では、ＪＶＭは、バイトコード命令がアンダフローを形成し得るか否かを決定する。アンダフローを形成し得る一般的な２つの浮動小数点オペレーションは乗算及び除算である。記述した実施例では、ＪＶＭは、特定の命令がアンダフローの問題を形成できないことを決定するとき、ステップ７０５に示すように、ＪＶＭはバイトコードをコンパイルすべく処理を続行する。

【００３７】命令がアンダフローの問題を潜在的に形成し得る場合、ＪＶＭは、どのようにアンダフローが検出され、訂正されるを決定するプロセスを開始する。前記のように、記述した実施例では、ＪＶＭはアンダフローを検出するための明示的チェック（即ち、インライン・コード）またはトラップを使用し得る。別の実施例では、前記の方法に代えて、または前記の方法に加えて、

アンダフローを検出する別の方法を使用できる。

【００３８】ステップ７０７では、バーチャル・マシンは、アンダフローを検出するためのトラップに関連するカウンタが所定の閾値を越えたか否かをチェックする。トラップの一部として、各トラッピング命令に関連するカウンタをインクリメントするための命令が含まれている。任意のカウンタがある閾値を越えた場合、バイトコード・トランスレータを再呼び出しするための命令が更に含まれている。カウンタは、特定のバイトコード（この例では、浮動小数点命令）をどの方法でコンパイルするのかを決定するためにチェックできる情報、即ち、プロフィール・データの１つの例である。図７に示すように、カウンタ及びこれに類する情報６０７は、ネイティブ命令セットと一緒に維持可能である。別の実施例では、ＪＶＭがどの方法でバイトコードをコンパイルするために使用されるべきかを決定するために、カウンタに代えて、またはカウンタと一緒に、タイマなどの別の種類のデータを使用できる。

【００３９】ＪＶＭ上でのジャバ・クラス・ファイルの１回の実行中に複数回実施される特定の浮動小数点オペレーションを実行するために、特定の方法が使用される度に、カウンタが更新され得る。例えば、特定の命令を条件付きループ内に有することによって、カウンタの更新を行い得る。トラップを使用して訂正するアンダフローを特定の浮動小数点オペレーションが引き起こす度に、カウンタはインクリメントされる。図６に示すステップ５０９で説明した“第１の方法”は、命令をコンパイルするトラップ法に該当し得る。記述した実施例では、一般的に採用される実行のパスにおけるオペレーションの数を減少させることが好ましいため、第１の方法でコンパイルする際、カウンタを避けることが望ましい。図８に示す特定の実施例では、コンパイルする（そして、アンダフローを潜在的に形成する）特定の浮動小数点オペレーションのためのカウンタが閾値へ達していない場合、ＪＶＭは第１の方法（この例では、命令をコンパイルするトラップ法）の使用を継続する。カウンタが閾値を越えている場合、“第２の方法”でのリコンパイレーションを行うために、フラグをこの命令／方法に対して立てる。前記のように、各トラッピング命令に関連するカウンタをインクリメントするための命令がトラップの一部として含まれている。

【００４０】トラップ法を使用した命令のコンパイルにおける第１工程は、ステップ７０９に示すように、トラップ・ハンドラがセットアップされているか否かを決定することである。ジャバ・クラス・ファイルがトラップを初めて呼び出した際、トラップ・ハンドラが形成される。トラップ・ハンドラを必要とするコードをコンパイルすることを、コンパイラが初めて決定した際（トラップ・ハンドラはこれ以前に必要となり得ない）、トラップ・ハンドラが（ＪＶＭによって）セットアップされる。記述した実施例及び始どのジャバ・プログラムでは、１つのトラップ・ハンドラがプログラム内の各スレッドに対して存在する。スレッドの詳細な説明は“ジャバ言語詳説”に開示されており、この文献の内容はこの開示をもって本明細書に開示したものとする。トラップ・ハンドラを形成する場合、これはステップ７１１で行われる。トラップ・ハンドラが特定のスレッドに対して既にセットアップされている場合、ステップ７１３に示すように、ＪＶＭはトラップ法を使用して命令をコンパイルする。トラップ・ハンドラを使用して命令をコンパイルするこのプロセスは、図９において更に詳述する。コンパイルを終えると、ＪＶＭは他のバイトコードがジャバ・クラス・ファイル内に存在するか否かを見るためにステップ７１５においてチェックする。存在する場合、ＪＶＭはステップ７０１へ戻り、プロセスを繰り返す。

【００４１】ステップ７０７では、ＪＶＭは、カウンタが所定値を超えたか否か（即ち、命令が特定の方法で所定回数を超えて実行されたか否か）をチェックする。超えている場合、ＪＶＭはバイトコードを次の方法でコンパイルする。この例では、その方法は、浮動小数点アンダフローを検出し、訂正するために、ステップ７１７に示すように、明示的チェック（インライン・コード）を使用する。別の実施例では、バイトコード・トランスレータがコンパイルされたバイトコードを特定の方法で実行し続けるべきか否かを決定するために、カウンタ以外の基準を使用することができる。バイトコード命令をコンパイルする明示的チェック法は図１０において詳述する。ステップ７１９では、バーチャル・マシンは明示的チェックと一緒に使用するタイマをセットする。その時間は、明示的チェック法を使用した時間的な長さを測定するために、使用される。次いで、ステップ７１５では、バーチャル・マシンはそれ以上のバイトコード が存在するか否かをチェックする。何も存在しない場合、ジャバ・クラス・ファイル内のバイトコードをコンパイルするプロセスは完了する。

【００４２】図９は浮動小数点命令からのアンダフローを処理するために、トラップを使用する図８のステップ７１３のプロセスの詳細を示すフローチャートである。ステップ８０１では、ＪＶＭは、ジャバ・クラス・ファイル内のどのバイトコード命令がトラップを呼び出しているかを決定する。バーチャル・マシンは、どの命令がトラップを呼び出しているかを決定すると、バーチャル・マシンは浮動小数点オペレーションに関連するカウンタをステップ８０３でインクリメントする。図７で述べたように、カウンタをネイティブ命令と一緒に維持し得る。カウンタがインクリメントされると、バーチャル・マシンはカウンタの値をステップ８０５でチェックする。カウンタが閾値を越えている場合、ステップ８０７において、浮動小数点命令を含むモジュールに対して、

リコンパイルするためにフラグを立てる。

【００４３】記述した実施例では、モジュールを即座にリコンパイルしない。その代わりに、そのリソース及びアクティビティのレベルに基づいてバーチャル・マシンによって決定された時点において、モジュールは、リコンパイルするために、キューへ加えられる。別の実施例では、モジュールを即座にまたは設定時間にコンパイルし得る。モジュールをリコンパイルする実際の時間とは無関係に、記述した実施例では、カウンタに基づいて、バーチャル・マシンはリコンパイルすることを決定する。別の実施例では、バーチャル・マシンは図７で述べたネイティブ命令と一緒に格納可能なダイナミックに形成されたプロフィール・データから得られた別の印を使用できる。ステップ８０７で、モジュールに対して、リコンパイルするために、フラグを立てるか、別の方法でマークすると、バーチャル・マシンは、ジャバ・クラス・ファイル内にそれ以上のバイトコードが存在するか否かをチェックするために、図８のステップ７１５へ戻る。カウンタがステップ８０５で所定値を超えていない場合、ＪＶＭは、ステップ８０９で、浮動小数点アンダフローを処理するために、トラップを使用することによって処理を継続する。次いで、バーチャル・マシンは図８のステップ７１５へ戻る。

【００４４】図１０は図８のステップ７１７で述べた浮動小数点アンダフローを検出し、訂正するための明示的チェック・ルーチンを示すフローチャートである。前記のように、明示的チェックは、浮動小数点アンダフローを検出し、訂正するために、ＪＶＭによってジャバ・クラス・ファイルから形成されたネイティブ命令セット内へ挿入されたコードである。図１０は、バーチャル・マシンが、明示的チェック法（明示的チェック法は図６のステップ５１１で述べた“次の方法”に該当し得る）でコンパイルする時、または図９のステップ８０７同様に、リコンパイルするために浮動小数点命令を含むモジュールに対してフラグを立てる時を、どのように決定するかを示す。ステップ９０１では、バーチャル・マシンは、浮動小数点アンダフローを訂正するために、明示的チェック法を使用する。前記のように、明示的チェックによってアンダフローが何回検出されたかを追跡するために、明示的カウンタを挿入し得るか、またはタイマを使用し得る。カウンタをこのパスで使用することによる１つの潜在的問題としては、カウンタが処理の期間で比較的コスト高となり得る点が挙げられる。別の実施例では、タイマ及びカウンタの組み合わせを使用可能である。次いで、バーチャル・マシンは明示的チェックを最初に使用した時点から何時間経過したかをチェックする。図８のステップ７１９において思い起こすと、記述した実施例では、命令を明示的チェックとしてコンパイルした後、バーチャル・マシンはタイマをセットする。所定時間を経過したか否かを決定するために、これと同じタイマをステップ９０３で使用する。記述した実施例では、所定時間を経過した場合、ステップ９０５において、ＪＶＭはコンパイルするためにモジュールに対してフラグを立てるか、またはマークをつける。

【００４５】１つの実施例では、特定の時間を経過した後、ＪＶＭはバイトコードをリコンパイルする。これは、現在コンパイルしている方法をデフォルトの方法へリセットすることによって、ジャバ・クラス・ファイルの実行が新たな状況へ適合せずに潜在的に非効率的になることを防止するために、行われる。バーチャル・マシンが決定した時点で、リコンパイルするためにモジュールに対してフラグを立て、キューへ加えると、この特定の浮動小数点命令に対応するカウンタ及び他のプロフィール・データをリセットまたはリフレッシュし、これによって、新しいプロフィール情報を維持できる。カウンタをステップ９０７でリセットする。次いで、バーチャル・マシンは図８のステップ７１９へ戻る。

【００４６】本発明はコンピュータ・システム内に格納された情報を使用する様々なコンピュータ実装オペレーションを使用する。これらのオペレーションは物理量の物理操作を必要とするオペレーションを含む（但し、同オペレーションに限定されない）。一般的に、必ずしも必要でないが、これらの量は格納、転送、結合、比較及び他の操作が可能な電気信号または磁気信号の形態をなす。本発明の一部を構成するここで記述するオペレーションは、有用なマシン・オペレーションである。実施する操作は形成、識別、実行、決定、比較、実行、ダウンロードまたは検出等の用語で示されることが多い。主に共通の用法を得る理由で、これらの電気信号または磁気信号をビット、値、エレメント、変数、キャラクター等として示すと時に都合が良い。しかし、これらの用語またはこれらに類似する用語の全ては適切な物理量に関連づけるべきであり、かつ、これらの量に適用された都合の良いラベルにすぎない点を覚えておく必要がある。

【００４７】更に、本発明は前記のオペレーションを実施するためのデバイス、システムまたは装置に関する。システムは要求された目的のために特別に構築可能であり、または、システムは、そのコンピュータに格納されたコンピュータ・プログラムによって選択的に作動または構成される汎用コンピュータとすることが可能である。前記のプロセスは特定のコンピュータまたは他のコンピューティング装置に本質的には関連しない。特に、様々な汎用コンピュータを、ここで開示されていることに基づいて記述されたプログラムと併用してよく、あるいは、これに代えて、要求されたオペレーションを実施するために、より特別なコンピュータ・システムを形成することはより都合が良い。

【００４８】図１１は本発明の一実施例に従う処理の実施に適した汎用コンピュータ・システム１０００のブロック図である。例えば、ＪＶＭ３０７、バーチャル・マ

シン３１１またはバイトコード・コンパイラ３０３を汎用コンピュータ・システム１０００上で実行できる。図１１は汎用コンピュータ・システムの一実施例を示す。本発明の処理を実施するために、他のコンピュータ・システム・アーキテクチャ及び構成を使用することができる。以下に記述する様々なサブシステムからなるコンピュータ・システム１０００は、少なくとも１つのマイクロプロセッサ・サブシステム（中央処理装置、即ち、ＣＰＵとも称される）１００２を含む。即ち、ＣＰＵ１００２はシングルチップ・プロセッサまたはマルチプル・プロセッサによって実現し得る。ＣＰＵ１００２はコンピュータ・システム１０００のオペレーションを制御する汎用デジタル・プロセッサである。メモリから取り出した命令を使用して、ＣＰＵ１００２は入力情報の受信及び操作と、出力デバイス上での情報の出力及び表示とを制御する。

【００４９】ＣＰＵ１００２は、メモリ・バス１００８を介して、一般的にランダム・アクセス・メモリ（ＲＡＭ）からなる第１の一次ストレージ１００４に双方向接続され、一般的にリード・オンリ・メモリ（ＲＯＭ）からなる第２の一次ストレージ領域１００６に単方向接続されている。当該技術分野でよく知られているように、一次ストレージ１００４は汎用ストレージ領域及び作業メモリとして使用可能であり、さらには入力データ及び処理済みデータを格納するためにも使用できる。更に、ＣＰＵ１００２上で処理されるプロセスのためのデータ及び命令を格納する以外に、一次ストレージ１００４はプログラミング命令及びデータを格納可能であり、そして、一般的に、データ及び命令を、メモリ・バス１００８の間を双方向で高速転送するために、使用される。同様に、当該技術分野でよく知られているように、一次ストレージ１００６は、一般的に、ＣＰＵ１００２がその機能を果たすために使用する基本オペレーティング命令、プログラム・コード、データ及びオブジェクトを含む。一次ストレージ・デバイス１００４，１００６は、例えば、データ・アクセスが双方向または単方向のいずれを必要とするかに依存して、以下に詳述する適切なコンピュータ読み取り可能ストレージ媒体を含み得る。ＣＰＵ１００２は、キャッシュ・メモリ１０１０において、頻繁に必要となるデータを超高速で直接取り出し、そして、格納できる。

【００５０】取り外し可能大容量ストレージ・デバイス１０１２はコンピュータ・システム１０００のための別のデータ・ストレージ容量を提供し、ペリフェラル・バス１０１４を介してＣＰＵ１００２に双方向または単方向のいずれかで接続されている。例えば、ＣＤ－ＲＯＭとして知られている特定の取り外し可能大容量ストレージ・デバイスは、一般的にデータを単方向でＣＰＵ１００２へ送信する。その一方、フロッピー・ディスクはデータを双方向でＣＰＵ１００２へ送信し得る。ストレージ１０１２は、磁気テープ、フラッシュ・メモリ、搬送波に具現化された信号、スマート・カード、ポータブル大容量ストレージ・デバイス及び他のストレージ・デバイス等のコンピュータ読み取り可能媒体を更に含み得る。また、固定大容量ストレージ１０１６は、別のデータ・ストレージ容量を提供し、ペリフェラル・バス１０１４を介してＣＰＵ１００２に双方向で接続されている。一般的に、これらの媒体へのアクセスは一次ストレージ１００４，１００６へのアクセスより遅い。大容量ストレージ１０１２，１０１６は、一般的に、ＣＰＵ１００２が頻繁に使用しない他のプログラミング命令及びデータ等を格納する。必要に応じて、大容量ストレージ１０１２，１０１６内に保持された情報は、一次ストレージ１００４（例：ＲＡＭ）の一部を構成するバーチャル・メモリとして標準的に組込み可能である。

【００５１】ストレージ・サブシステムへのＣＰＵ１００２のアクセスを提供する以外に、ペリフェラル・バス１０１４は、同様に、他のサブシステム及びデバイスへのアクセスを提供するために使用される。記述した実施例では、これらは、ディスプレイ・モニタ１０１８及びアダプタ１０２０、プリンタ・デバイス１０２２、ネットワーク・インターフェース１０２４、補助入力／出力装置インターフェース１０２６、サウンド・カード１０２８及びスピーカ１０３０、並びに必要とされる他のサブシステムを含む。図示するように、ネットワーク接続を使用することにより、ネットワーク・インターフェース１０２４はＣＰＵ１００２を別のコンピュータ、コンピュータ・ネットワークまたは通信ネットワークへ接続可能にする。前記の方法のステップを実行するうえで、ネットワーク・インターフェース１０２４を通じることで、ＣＰＵ１００２が、オブジェクト、プログラム命令またはバイトコード命令などの情報を別のネットワーク内のコンピュータから受信するか、または情報を別のネットワーク内のコンピュータに出力することを意図している。ＣＰＵで実行する命令のシーケンスとしてしばしば表される情報は、例えば、搬送波に具現化されたコンピュータ・データ信号の形態で別のネットワークに対して送受信可能である。インターフェース・カードまたはこれに類似するデバイスと、ＣＰＵ１００２によって実行される適切なソフトウェアとは、コンピュータ・システム１０００を外部ネットワークへ接続し、標準プロトコルに従ってデータを転送するために使用できる。即ち、本発明で具現化される方法はＣＰＵ１００２上で単独で実行してよいし、または、処理の一部を共有する遠隔ＣＰＵと協働することにより、インターネット、イントラネットワーク若しくはローカル・エリア・ネットワーク等のネットワークを通じて実行してよい。また、別の大容量ストレージ・デバイス（図示せず）をネットワーク・インターフェース１０２４を通じてＣＰＵ１００２へ接続してよい。

【００５２】補助入力／出力装置インターフェース１０２６は、ＣＰＵ１００２に他のデバイスにデータを送信させ、より一般的に、そのデバイスからのデータを受信させ得る汎用及びカスタム・インターフェースを表す。キーボード１０３６またはポインタ・デバイス１０３８からの入力を受信し、さらにはデコードしたシンボルをキーボード１０３６またはポインタ・デバイス１０３８からＣＰＵ１００２へ送信するために、キーボード・コントローラ１０３２がローカル・バス１０３４を通じてＣＰＵ１００２へ接続されている。ポインタ・デバイスは、マウス、スタイラス、トラック・ボールまたはタブレットであってよく、そして、グラフィカル・ユーザ・インターフェースとの相互作用に有用である。

【００５３】加えて、本発明の実施例は、さらに、様々なコンピュータ実行オペレーションを実施するためのプログラム・コードを含むコンピュータ読み取り可能媒体を有するコンピュータ・ストレージ製品に関する。コンピュータ読み取り可能媒体は、コンピュータ・システムによって後からの読み取りが可能なデータを格納し得る任意のデータ・ストレージ・デバイスである。コンピュータ読み取り可能媒体の例としては、ハード・ディスクと、フロッピー・ディスクと、特定用途向け集積回路（ＡＳＩＣ）またはプログラム可能論理回路（ＰＬＤ）などの特別に形成されたハードウェア・デバイスと、を含めた前記の全ての媒体が挙げられる（但し、これらに限定されない）。また、コンピュータ読み取り可能媒体は、搬送波に具現化されたデータ信号として結合コンピュータ・システムのネットワーク上に分散させ得る。従って、コンピュータ読み取り可能コードは分散した形態で格納及び実行できる。

【００５４】前記のハードウェア・エレメント及びソフトウェア・エレメントが標準的な設計及び構成を有することを当業者は理解し得る。本発明を使用するのに適した他のコンピュータ・システムは別のサブシステムまたは少数のサブシステムを含み得る。更に、メモリ・バス１００８、ペリフェラル・バス１０１４及びローカル・バス１０３４はサブシステムをリンクするのに役立つ任意の相互接続方式の実例である。例えば、ローカル・バスはＣＰＵを固定大容量ストレージ１０１６及びディスプレイ・アダプタ１０２０へ接続するために使用可能である。しかし、図１１に示すコンピュータ・システムは本発明を使用するのに適したコンピュータ・システムの例である。サブシステムの別の構成を有する他のコンピュータ・アーキテクチャを利用し得る。

【００５５】以上、理解を容易にする目的で、本発明をある程度詳しく説明したが、本発明の請求の範囲内において、特定の変更及び修正を実施しても良い。例えば、トラップ法及び明示的インライン・チェック法を浮動小数点アンダフローに関連して詳述したが、アンダフローを検出する他のツールも使用し、本発明に組み入れることができる。別の例では、命令をコンパイルする２つの方法を開示したが、プログラムをコンパイルする更に多くの方法を利用可能な場合、本発明の方法及び装置はこれら２つより多い方法に対応可能である。更に、本発明を浮動小数点アンダフロー・オペレーションを使用して説明したが、プラットフォーム固有のバリエーションに起因する問題の解決に使用するために、本発明はインプリメンテーションをインテリジェントで、ダイナミックに選択できる。浮動小数点アンダフローは、この種の問題の１つに過ぎない。更に、本発明の方法及び装置の両方をインプリメントするための代わりの方法が存在することを認識する必要がある。従って、本実施例は例示を目的とするものであって、限定を目的としない。更に、本発明はここで与えられた詳細部分に限定されることなく、添付の請求の範囲及びそれに等価なものの範囲内で変更できる。

【図面の簡単な説明】

【図１】従来技術で知られているような倍精度浮動小数点の一般的なフォーマットと拡張精度浮動小数点のフォーマットを示すブロック図である。

【図２】従来技術で知られているような浮動小数点アンダフローを検出する２つの方法を示す図である。

【図３】ジャバ・ソース・コードを含むジャバ（商標名）プログラムを特定のプラットフォーム、即ち、コンピュータ上で実行されるネイティブ・コードに変換することを示すブロック／プロセス図である。

【図４】図１１のコンピュータ・システム１０００によってサポートされているバーチャル・マシン３０７を示す図である。

【図５】ネイティブ命令の異なるバージョンがジャバ・プログラムからどのように形成されるのかを示すブロック図である。

【図６】本発明の一実施例に従う、ジャバ・バイトコードをコンパイルするジャバ・バーチャル・マシンのプロセスを示すフローチャートである。

【図７】本発明の一実施例に従う、ダイナミックに形成されたプロフィール・データを含むネイティブ命令がどのように形成されるのかを示すブロック図である。

【図８】本発明の一実施例に従う、浮動小数点オペレーションをコンパイルするとともに、アンダフローが発生した場合、このアンダフローを訂正する方法を決定するジャバ・バーチャル・マシンを示すフローチャートである。

【図９】図８のステップ７１３に示す浮動小数点命令からのアンダフローを処理するトラップ・ルーチンを使用するプロセスをより詳細を示すフローチャートである。

【図１０】図８のステップ７１７に示す浮動小数点アンダフローを検出し、訂正するための明示的チェック・ルーチンを示すフローチャートである。

【図１１】本発明の一実施例を実現することに適した一

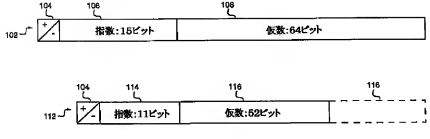
般的なコンピュータ・システムのブロック図である。

【符号の説明】

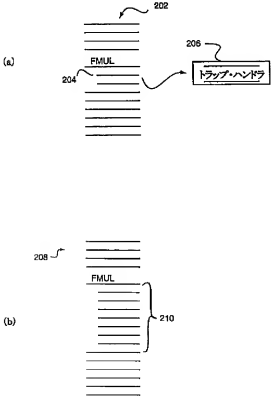
１０００…コンピュータ・システム
１００２…ＣＰＵ
１００４…第１の一次ストレージ
１００６…第２の一次ストレージ領域
１００８…メモリ・バス
１０１０…キャッシュ・メモリ
１０１２…取り外し可能大容量ストレージ・デバイス
１０１４…ペリフェラル・バス
１０１６…固定大容量ストレージ

＊１０１８…ディスプレイ・モニタ
１０２０…アダプタ
１０２２…プリンタ・デバイス
１０２４…ネットワーク・インターフェース
１０２６…補助入力／出力装置インターフェース
１０２８…サウンド・カード
１０３０…スピーカ
１０３２…キーボード・コントローラ
１０３４…ローカル・バス
１０３６…キーボード
１０３８…ポインタ・デバイス

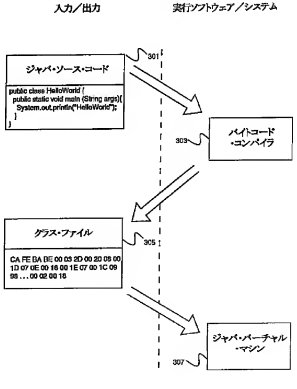
【図１】



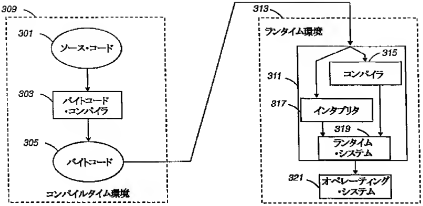
【図２】



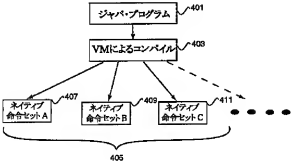
【図３】



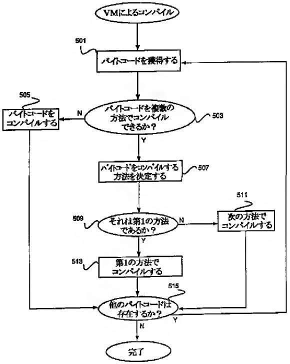
【図４】


309
301
ソース・コード
303
バイトコード
・コンパイラ
305
バイトコード
コンパイルタイム環境
313
ランタイム環境
315
311
コンパイラ
317
インタプリタ
319
ランタイム
・システム
321
オペレーティング
・システム

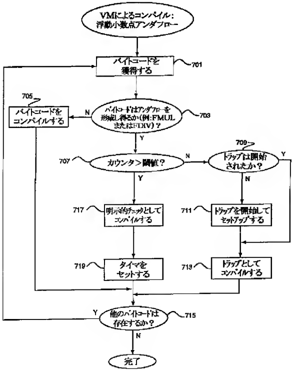
【図５】


ジャバ・プログラム
401
VMによるコンパイル
403
ネイティブ
命令セットA
407
ネイティブ
命令セットB
409
ネイティブ
命令セットC
411
405

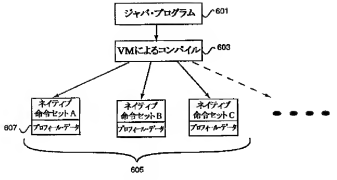
【図６】


VMによるコンパイル
501
バイトコードを獲得する
505
バイトコードを
コンパイルする
N
バイトコードを複数の
方法でコンパイル
できるか？
503
Y
バイトコードをコンパイルする
方法を決定する
507
511
509
それは第1の方法
であるか？
N
次の方法で
コンパイルする
Y
513
第1の方法で
コンパイルする
515
他のバイトコードは
存在するか？
Y
N
完了

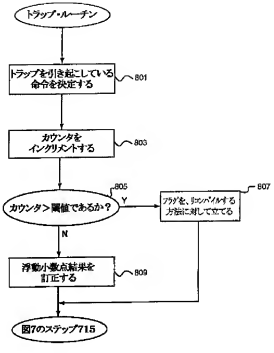
【図８】


VMによるコンパイル：
浮動小数点アンダフロー
バイトコードを
獲得する
701
705
バイトコードを
コンパイルする
N
バイトコードはアンダフローを
形成し得るか(例:FMUL
またはFDIV)？
703
Y
709
707
カウンタ＞閾値？
N
トラップは開始
されたか？
Y
Y
N
717
明示的チェックとして
コンパイルする
711
トラップを開始して
セットアップする
719
タイマを
セットする
713
トラップとして
コンパイルする
Y
他のバイトコードは
存在するか？
715
N
完了

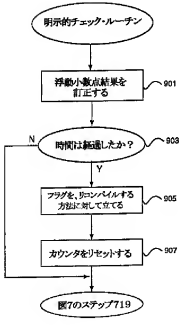
【図７】


ジャバ・プログラム
601
VMによるコンパイル
603
ネイティブ命令セットA プロフィールデータ
ネイティブ命令セットB プロフィールデータ
ネイティブ命令セットC プロフィールデータ
607
605

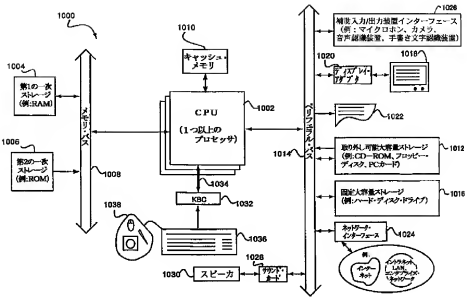
【図９】


トラップ・ルーチン
トラップを引き起こしている命令を決定する
801
カウンタをインクリメントする
803
カウンタ＞閾値であるか？
805
Y
N
フラグを、コンパイルする方法に対して立てる
807
浮動小数点結果を訂正する
809
図7のステップ715

【図１０】


明示的チェック・ルーチン
浮動小数点結果を訂正する
901
時間は経過したか？
903
N
Y
フラグを、コンパイルする方法に対して立てる
905
カウンタをリセットする
907
図7のステップ719

【図１１】



フロントページの続き

(71)出願人　591064003
901　SAN　ANTONIO　ROAD
PALO　ALTO，CA　94303，U．S．A．

(72)発明者　ピーター・ビー．・ケスラー
アメリカ合衆国　カリフォルニア州94306　パロ・アルト，ロス・ロブレス・アベニュー，769

【外国語明細書】

1 Title of Invention

METHOD AND APPARATUS FOR SELECTING WAYS TO COMPILE AT RUNTIME

2 Claims

1. A method of determining at runtime how to compile bytecode instructions associated with a computer program, the method comprising:

retrieving a bytecode instruction associated with the computer program that can be compiled in a plurality of ways;

compiling the bytecode instruction a first way;

determining at runtime that a second way of compiling the bytecode instruction is desirable; and

recompiling the bytecode instruction the second way.

2. A method as recited in claim 1, further comprising placing a method containing the bytecode instruction to be recompiled in a queue.

3. A method as recited in claim 1, further comprising determining at runtime that the bytecode instruction can be compiled in a plurality of ways.

4. A method as recited in claim 1, further comprising examining dynamically changing efficiencies of the computer program.

5. A method as recited in claim 4, further comprising gathering, at runtime, specific data on a currently executing one of the plurality of ways the computer program can be compiled.

6. A method as recited in claim 1, further comprising recompiling the

bytecode instructions the second way when the second way is different from the first way.

7. A method of generating different sets of native instructions from a software program, the method comprising:

compiling the software program in a first way to create a first set of native instructions;

determining at runtime of the software program that it would be desirable to compile the software program in a different way; and

recompiling the software program in the different way to create a second set of native instructions whereby the second set of native instructions replaces the first set of native instructions.

8. A method as recited in claim 7, further comprising examining dynamically generated specific data to determine which way would be beneficial to recompile the software program.

9. A method as recited in claim 8, wherein the dynamically generated specific data includes a counter that stores the number of times a particular compilation way has been executed.

10. A method as recited in claim 7, further comprising determining whether the first set of native instructions should be replaced by the second set of native instructions to more efficiently process dynamically changing requirements of the software program.

11. A method as recited in claim 7, further comprising determining which specific instructions in the software program are to be recompiled and marking the specific instructions at runtime of the software program.

12. A method of executing a floating point operation in a system, the method comprising:

determining whether a floating point operation can create a floating point underflow;

checking a particular indicator to determine how many times a specific floating point operation has caused an underflow; and

compiling the specific floating point operation using a first way when the particular indicator meets predetermined criteria and otherwise compiling the specific floating point operation using a second way.

13. A method as recited in claim 12 further comprising dynamically generating and storing data on the first and second ways of compiling the specific floating point operation at runtime.

14. A method as recited in claim 12 wherein the first way is a trap routine and the second way is an explicit check, the explicit check implemented by inserting inline code into the program.

15. A method as recited in claim 14 further comprising:

checking whether a predetermined amount of time has passed from the time the specific floating point operation was compiled using an explicit check; and

marking a module including the specific floating point operation to be recompiled if the predetermined amount of time has passed.

16. A method as recited in claim 14 further including initializing the trap routine.

17. A method as recited in claim 14 further including setting a flag if the specific floating point operation is compiled using an explicit check.

18. A method as recited in claim 14 wherein the particular indicator is a counter that is incremented each time the floating point underflow is handled using a trap routine.

19. A method as recited in claim 14, further comprising:

determining which floating point operation caused the trap routine to execute;

incrementing a counter; and

marking a module including the specific floating point operation to be recompiled if the counter is above a predetermined value.

20. A method as recited in claim 19, further comprising placing the module in a recompile queue if the module is marked to be recompiled.

21. A method as recited in claim 19, further comprising resetting the counter associated with the method if the method has been marked to be recompiled.

22. A method of generating instructions for detecting floating point underflow, the method comprising:

determining whether an operation in a program can create a floating point underflow;

examining an underflow indicator to measure a tendency of a particular floating point operation for causing an underflow;

compiling the particular floating point operation using a trap routine

according to the tendency of the particular floating point operation fo
r causing an underflow; and

recompiling the floating point operation using an inline exp'cit chec
k according to the tendency of the particular floating point operation f
or causing an underflow.

23. A method as recited in claim 22, further comprising dynamica 'y g
enerating and storing data associated with the trap routine and the acti
ne explicit check at runtime.

24. A method as recited in claim 23, wherein the particular floating
point operation is contained in a module and further comprising placing
the module containing the floating point operation to be recompiled in a
recompile queue.

25. A computer program product for determining at runtime how to comp
ile bytecode instructions associated with a computer program, comprising
:

a computer code that retrieves a bytecode instruction associated with
the computer program that can be compiled in a plurality of ways;

a computer code that compiles the bytecode instruction a first way;

a computer code that determines at runtime that a second way of compi
ling the bytecode instruction is desirable;

a computer code that recompiles the bytecode instruction the second wa
y; and

a computer readable medium that stores the computer codes.

26. A computer program product for executing a floating point operat
or in a program, comprising:

a computer code that determines whether a floating point operation will create a floating point underflow;

a computer code that checks a particular indicator to determine how many times a particular floating point operation has caused an underflow;

a computer code that compiles the particular floating point operation using a first way when the particular indicator meets predetermined criteria and otherwise compiling the floating point operation using a second way; and

a computer readable medium that stores the computer codes.

27. A system for determining at runtime how to compile bytecode instructions associated with a computer program, the system comprising:

a bytecode retriever for retrieving a bytecode instruction associated with the software program that can be compiled in a plurality of ways;

a compiling module for compiling the bytecode instruction in one of a first way and a second way; and

an alternative compiling detector for determining at runtime that the second way of compiling the instruction is desirable.

28. A system as recited in claim 27, further comprising a module queue for holding one or more modules, each module containing the bytecode instruction to be recompiled.

29. A system as recited in claim 27, further comprising an efficiency analyzer examiner for examining dynamically changing efficiencies of the computer program.

30. A system as recited in claim 27, further comprising a runtime data collecter for gathering, at runtime, specific data on a currently run

ing one of the plurality of ways the computer program can be compiled.

31. A system for executing a floating point instruction on a processor, the system comprising:

an instruction evaluator for determining whether a floating point instruction can create a floating point underflow;

a floating point underflow indicator for keeping a count of the number of times a particular floating point operation has caused a floating point underflow; and

a compiler for compiling the particular floating point operation in a first way when the particular indicator meets predetermined criteria and otherwise compiling the particular floating point operation using a second way, wherein the floating point underflow indicator is checked to determine how many times the particular floating point operation has caused an underflow.

32. A system as recited in claim 31 further comprising a data generator for dynamically generating and storing data on the first and second ways of compiling the floating point operation at runtime.

33. A system for determining at runtime how to compile a program including instructions comprising:

one or more processors; and

a computer readable medium storing a program for execution by the one or more processors comprising:

computer code that retrieves an instruction in the program that can be compiled in a plurality of ways;

computer code that compiles the instruction in a first way;

computer code that determines at runtime that a second way of compiling

g the instruction is executed and

computer code that recompiles the instruction in the second way.

3 Detailed Description of Invention

BACKGROUND OF THE INVENTION

The present invention relates generally to the field of computer software and software portability. In particular, it relates to methods of recompiling a program according to platform-specific requirements

The Java (trademark) virtual machine (JVM) can be implemented on a variety of different computer architectures and can accommodate different specifications and standards stemming from different microprocessors. An example of such a standard is the extended precision floating point format utilized by some microprocessors when manipulating floating point numbers. One such microprocessor is Intel Corporation's IA-32 microprocessor architecture which uses the extended precision (80 bit) floating point calculations. Other processors typically use single (32 bit) or double (64 bit) precision floating point calculations.

A problem occurs when values computed in extended precision format are converted to single or double precision format. The example problem arises because the Java (trademark) language specifies that floating point operations must produce results with the range and precision specified by IEEE 754, incorporated herein by reference for all purposes, whereas the Intel IA 32 processors, made by the Intel Corporation of Santa Clara, California, produce results with greater range and precision. These wider results must be accurately rounded to IEEE 754 single and double precision format. On the IA 32 microprocessor there are at least two ways to implement such rounding, each with different costs (code size and execution speed). A static compiler (or a one-time dynamic compiler) needs to choose one implementation, and that choice will not be the best ch

ice in all circumstances. The problem is illustrated in FIG.

FIG. 1 is a block diagram describing a typical format of double precision floating point and the format of extended precision floating point.

Format 102 illustrates an extended precision floating point number format of the Intel IA-32 architecture, as opposed to double precision floating point format of the Intel IA-32. A sign bit 104 indicates whether the number is positive or negative. This is followed by bits 106 which represent an exponent value for representing an exponential value of the floating point number.

Bits 108 contain bits for holding a significand. The significand portion can hold up to 64 bits for representing the integer portion of the number. Thus, there are 80 (1+15+64) bits in the extended precision floating point format. Floating point operations are typically handled by a floating point unit. This unit can efficiently perform complicated operations involving floating point numbers by manipulating the significand and the exponent. As is well known in the art, representing floating point numbers as integers and exponents make calculations on floating point numbers significantly easier.

Also referred to in FIG. 1 is a double precision floating point format 112, described in IEEE 754. This format is similar in layout to that of the extended precision format, but differs in the number of bits in the exponent and significand fields. As mentioned above, the Intel IA-32 processor produces results in extended format. The example problem comes from the The Java (trademark) Language Specification by James Gosling, Bill Joy, and Guy Steele (ISBN 0 201 63451-1), which is incorporated herein by reference in its entirety, in which results need to be produced in IEEE 754 single or double format. Returning to format 102, sign bit 104 is the same as in the extended format, as opposed to double or single precision format. Exponent bits 114 have the same function as bits 1

06 but holds 11 bits as opposed to 15 bits in the extended format. S ignificand bits 116 hold 52 bits as opposed to 64 bits in the extended format. Thus, the double precision floating point format can hold 64 bits.

The difference in the significand lengths is referred to by the dashed area 118 in FIG. 1 (the 4 bit difference in exponent length is not illustrated similarly in the figure).

Problems arise from the Java language requiring a single or double format result when given an extended precision result from an IA-32 process or, for example, if the extended exponent is outside the range of single or double precision, overflow or underflow will occur. On the IA-32, overflow is handled by the hardware, but can be addressed by methods described in the present invention. Underflow is more difficult to treat since the significand can be shifted (to the right) to reduce the exponent. However, this shifting loses bits of the significand, and therefore precision of the result. Computing the correct, less precise, significand requires several instructions, and the operation is typically put in a separate subroutine to be invoked when needed. The example problem is not the correct rounding of the result, but the detection that the correction should occur.

FIGS. 2a and 2b illustrate two methods of detecting floating point underflow. In one method, referred to in FIG. 2a, program code 202 detects the problem by performing a trap using a trap handler 204 in the program code to call a trap routine 206, which corrects the problem. In another method, referred to in FIG. 2b, program code 208 includes code 210 for detecting the problem wherever floating points are used in a manner where the problem can potentially arise, such as with multiplication and division operations.

By utilizing trap handler 206, all operations will cease while the problem is being addressed. When a trap is invoked, the state of the machi

ne is stored before executing the trap routine, including the location o
f the instruction causing the trap. However, if the trap is not invoked
, there is no per-operation overhead, only a one-time overhead per thread for setting up the trap handler, that will then monitor all floating p
oint operations. On the other hand, program code 210 shows a technique of inserting code in the program to handle floating point underflow prob
lems. With this method, each operation that might cause an underflow pr
oblem is followed by inline code to check for the problem and invoke a s
ubroutine, if necessary, to produce the correctly rounded result. This method requires many unnecessary processor operations for each underflow that does not occur. However, when the problem is detected, it is solv
ed without having to temporarily stop all operations and having to save a process or thread context to handle a trap. As mentioned above, floating point underflow is one example of a problem having alternative solutions on a given platform. Other problems can occur in which several implementations for solving a particular problem are available to the Java virtual machine, with each implementation being more efficient in some cases.

Therefore, it would be desirable to choose intelligently and dynamically an implementation to use in solving a problem arising from platform-s
pecific variations. Using the floating point underflow problem as such an illustration, it would be desirable, for example, to detect and correct floating point underflow while reducing the amount of inline code nec
essary and avoiding the overhead of using a trap handler to dispatch a s
ubroutine to correct the problem. It would also be desirable to allow a dynamic runtime compiler to choose one implementation, and also modify its efficiency and change its implementation if desired.

SUMMARY OF THE INVENTION

According to the present invention, methods, apparatus, and computer p rogram products are disclosed for a virtual machine to determine how to compile bytecode associated with a program while the program is executi ng. In one aspect of the present invention, an instruction in the progra m that may be compiled at runtime in multiple ways is retrieved and comp iled in a particular way, typically the default way. The virtual machine may then determine at runtime that another way of compiling the instruc tion is more desirable and the bytecode instruction is then recompiled the other way.

In one embodiment, the executable code that contains the bytecode inst ruction to be recompiled is placed in a queue with other instructions th at are to be recompiled. The virtual machine examines any changing requ irements of the program that have developed as the program's execution w here the requirements are derived from profile data on each one of the m ultiple ways the program can be compiled. In another embodiment, the pa rticular bytecode instruction is recompiled by the virtual machine in a way that is different from the first or default way of compiling the ins truction.

In another aspect of the invention, a method of generating different s ets of executable instructions from a single program is provided. A pro gram is compiled a certain way at runtime, such as the default way, to create one set of bytecode instructions. A virtual machine then determi nes at runtime that it would be desirable to compile the program in a di fferent way and does so creating a different set of native machine code which replaces the first set.

In one embodiment, the virtual machine examines dynamically generated profile data on each of the ways a program can be executed to determine which way to recompile the program. The profile data includes a counter that stores the number of times the program was executed in a particula

r way. The virtual machine determines whether a particular se of se. e instructions should be replaced by another set of native instructions to more efficiently process dynamically changing requirements of the pro gram.

In another aspect of the present invention, a system for executing a f loating point instruction in a program is described. The system determ nes whether a particular instruction can create a floating point underfl ow and checks an indicator to determine how many times a floating point operation has caused an underflow. The floating point operation is comp iled at runtime by a virtual machine one way if the indicator is below a predetermined value and is runtime compiled another way if the indicato r is above the predetermined value.

In yet another aspect of the present invention, a method of generating instructions for detecting floating point underflow using either a trap routine or an explicit check is described. It is determined whether an operation in a program can create a floating point underflow and checks a counter to determine how many times a particular floating point opera tion caused an underflow. The operation is then runtime compiled using a trap routine if the counter is below a predetermined value and recomp iled using an inline explicit check if the counter is above a predeterm ed value. Data associated with the trap routine and the inline explic t check are dynamically generated and stored at runtime.

The invention will be better understood by reference to the following description taken in conjunction with the accompanying drawings

DETAILED DESCRIPTION

Reference will now be made in detail to a specific embodiment of the I nvention. An example of this embodiment is illustrated in the accompany ing drawings. While the invention will be described in conjunction with

a specific embodiment. It will be understood that it is not intended to limit the invention to one embodiment. To the contrary, it is intended to cover alternatives, modifications, and equivalents as may be included within the spirit and scope of the invention as defined by the appended claims.

The present invention addresses choosing between alternative implementations of performing certain types of operations on a given architecture. Conventional methods typically generate code once, either statically at compile-time, or dynamically at runtime. The invention described and claimed herein allows a virtual machine to choose at runtime, which of a plurality of possible code segments will be generated by the runtime compiler based on runtime performance data. It allows a dynamic compiler to choose one implementation while monitoring its efficiency and changing the implementation if desired.

As described above, there are two common ways to detect and fix floating point underflow. One is to use a trap method which involves using code that is fast and short, but requires holding normal program execution while the trap is handled. The other method involves inserting code in the program to detect underflow after each floating point operation, which requires that the code be executed each time, but allows a program to proceed sequentially.

The underflow problem arises when it is necessary to store an extended floating point format result in a single or double floating point format. This can occur, for example, when a result is computed on an Intel architecture computer and then stored in single or double precision format. More specifically, the problem occurs when performing calculations using very small numbers, such as when the exponent of the result is less than the smallest exponent that can be represented in the destination (IEEE 754 single or double precision format), and the significand of the

result of rounding the exact result. In order to store the [illegible] representation in the destination, one must know that the significand of the exact (and therefore rounded), and which way it was rounded and why. This detection, and correction, is important to maintain the accuracy of very small numbers, e.g., to measure the rate at which a value is approaching zero.

FIG. 3a is a block diagram showing the inputs/outputs and the executing software/systems involved in creating native instructions from Java source code in accordance with one embodiment of the present invention. In other embodiments, the present invention can be implemented with a virtual machine for another language or with class files other than Java class files. Beginning with the left side of the diagram, the first input is Java source code 301 written in the Java (trademark) programming language developed by Sun Microsystems of Mountain View, California. Java source code 301 is input to a bytecode compiler 303. Bytecode compiler 303 is essentially a program that compiles source code 301 into bytecodes. Bytecodes are contained in one or more Java class files 305. Java class file 305 is portable in that it can execute on any computer that has a Java virtual machine (JVM). Components of a virtual machine are shown in greater detail in FIG. 3B. Java class file 305 is input to a JVM 307. JVM 307 can be on any computer and thus need not be on the same computer that has bytecode compiler 303. JVM 307 can operate in one of several roles, such as an interpreter or a compiler. If it operates as a compiler, it can further operate as a "just in time" (JIT) compiler or as an adaptive compiler. When acting as an interpreter, the JVM 307 interprets each bytecode instruction contained in Java class file 305.

FIG. 3b is a diagrammatic representation of virtual machine 311 such as JVM 307, that can be supported by computer system 1000 of FIG. 1A described below. As mentioned above, when a computer program, e.g., a progr

am written in the Java (trademark) programming language. A program f rom source to bytecodes, source code 301 is provided to a bytecode compi ler 303 within a compile-time environment 303. Bytecode compiler 303 tr anslates source code 301 into bytecodes 305. In general, source code 30 1 is translated into bytecodes 305 at the time source code 301 is create d by a software developer.

Bytecodes 305 can generally be reproduced, downloaded, or otherwise di stributed through a network, e.g., through network interface 1024 of FIG . 10, or stored on a storage device such as primary storage 1064 of FIG . 10. In the described embodiment, bytecodes 305 are platform independen t. That is, bytecodes 305 may be executed on substantially any computer system that is running a suitable virtual machine 311. Native instruct ions formed by compiling bytecodes may be retained for later use by the JVM. In this way the cost of the translation are amortized over multipl e executions to provide a speed advantage for native code over interpret ed code. By way of example, in a Java (trademark) environment, bytecode s 305 can be executed on a computer system that is running a JVM.

Bytecodes 305 are provided to a runtime environment 312 which includes virtual machine 311. Runtime environment 312 can generally be executed using a processor such as CPU 1002 of FIG. 10. Virtual machine 311 in cludes a compiler 315, an interpreter 317, and a runtime system 319. Byt ecodes 305 can generally be provided either to compiler 315 or interpret er 317.

When bytecodes 305 are provided to compiler 315, methods contained in bytecodes 305 are compiled into machine instructions (see, e.g., FIG. 4 On the other hand, when bytecodes 305 are provided to interpreter 317, bytecodes 305 are read into interpreter 317 one bytecode at a time. In terpreter 317 then performs the operation defined by each bytecode as ea ch bytecode is read into interpreter 317. In general, interpreter 317 p

processes bytecodes 305 and performs operations associated with bytecodes 305 substantially continuously.

When a method is called from an operating system 321, it is determined that the method is to be invoked as an interpreted method. runtime system 319 can obtain the method from interpreter 317. If, on the other hand, it is determined that the method is to be invoked as a compiled method, runtime system 319 activates compiler 313. Compiler 313 then generates native machine instructions from bytecodes 305, and executes these machine-language instructions. In general, the machine-language instructions are discarded when virtual machine 311 terminates. The operation of virtual machines or, more particularly, Java (trademark) virtual machines, is described in more detail in The Java (trademark) Virtual Machine Specification by Tim Lindholm and Frank Yellin (ISBN 0-201-63452-X), which is incorporated herein by reference in its entirety.

As described earlier, instructions in a Java program can sometimes be compiled in more than one way. Following the earlier example, a floating point operation that can potentially cause underflow, such as a multiplication (FMUL) or division (FDIV) operation, can be compiled in at least two different ways: with an explicit check or a trap. FIG. 4 is a flow diagram describing how different versions of native instructions can be generated from a Java program. A Java program 401 (input 301 in FIGS. 3a and 3b), after being compiled to a bytecode compiler into Java class files, program 401 is runtime compiled from the bytecodes to native machine instructions by a JVM at block 403 (system 307 in FIG. 3a). A JVM is used for purposes of illustration only. As is known to a person skilled in the art, a virtual machine applies to general translation from a given input representation to a native instruction set where there is a choice of implementation. A method of compilation of the Java class files by a JVM is described in FIG. 5 below.

As mentioned earlier, a JVM can assume one of two roles: interpreting the Java bytecodes contained in the class files, or compiling the class files thereby creating native instruction sets which run on the same computer that has the JVM (i.e., they are not portable). Thus, with respect to the JVM performing as a compiler, a variety of native instruction sets can result from the same Java program depending on how the JVM compiles the bytecodes as referred to in blocks 405. Using the floating point operation as an example, native instructions 407 can contain explicit checks (i.e. inline) in all its FMULs or FDIVs, whereas native instructions 409 can contain only traps for the same floating point operations, or native instructions 411 can contain a combination of both.

It is notable here that FIG. 4 is not depicting a JVM deciding at runtime which compilation route to take (i.e., whether to compile the Java source code, interpret the code, or perform other operations with regard to how or when to execute the source code). Instead, FIG. 4 illustrates that if the compilation route taken by the JVM is to compile the code at runtime it can do so in different "ways," thereby creating different sets of native instructions. This process is described in detail with regard to FIG. 5.

FIG. 5 is a flowchart describing a process of a Java virtual machine compiling Java bytecodes into native machine instructions in accordance with one embodiment of the present invention. At step 501 the JVM receives one or more bytecode instructions from a Java class file. At step 503 the JVM determines whether a particular instruction can be compiled in more than one way. A specific example of a bytecode instruction that can be compiled in more than one way is described in greater detail in FIG. 7. If the JVM determines that the instruction can be compiled in only one way, such as an IADD or LSUB operation, the JVM compiles the bytecode in step 505. The JVM determines whether there are any remaining by

tecodes at step 513 after it compiles the previously retrieved bytecode.

If the JVM determines that there are multiple ways to compile the byte code, it proceeds to determine which way it will compile the bytecode at step 507. In the described embodiment, a mechanism is used by the JVM to make this determination, as described in greater detail in FIG. 6. The mechanism involves using dynamically generated profile information on each of the different ways a bytecode instruction can be compiled. At step 509 the JVM determines whether to compile the bytecode using a default way. The default way typically would be the way the runtime compiler writer believed, after considering the options available at the time would ordinarily be the most efficient or logical way.

If, at step 509, the JVM determines that the bytecode should be compiled the first way, it does so at step 513 and produces a first native instruction set, such as native instruction set A in FIG. 4. It then determines whether there are any other bytecodes in the class file at step 515. If there are, it returns to step 501 to retrieve the next bytecode.

If there are no more bytecodes the process is complete.

If, at step 509, the JVM determines that the way to compile the bytecode is not the first or default way, the JVM compiles the bytecode instruction using another compilation technique at step 511. It then proceeds as from step 513 and determines whether there are any remaining bytecodes to be compiled in step 515. For simplicity, only two different ways are referred to in FIG. 5, but the invention may be advantageously applied to three or more ways of compiling bytecodes.

FIG. 6 is a block diagram illustrating how native machine instructions containing dynamically generated profile data are generated in accordance with one embodiment of the present invention. FIG. 6 is similar to FIG. 4 except that it includes information about each of the different ways a bytecode instruction can be compiled by a JVM. The information is

generated if it is determined that there are multiple ways to compile the bytecode into native instructions as referred to in step 505 of FIG. 5. At the top of FIG. 6 is a Java program 601 which, after being compiled into bytecodes by a bytecode compiler, is input to a Java virtual machine 603. The JVM can then output several different native instruction sets 605 based on the different ways bytecodes can be compiled into native instructions. The native instruction sets 605 can also include a data space for storing dynamically gathered data 607 collected at runtime. This information may include profile information such as counters, timing data, and other information relating to the efficiency of the particular way the bytecodes are compiled.

The dynamically gathered data may also be stored with the native instructions and updated while the bytecodes are compiled by the JVM. In one embodiment, the JVM examines this information to determine which way the bytecode should be compiled, as first described in step 507 of FIG. 5.

The dynamic profile data can be used by the JVM to determine whether, for example, a particular way of compiling continues to be efficient, how many times the bytecode has been executed that way, or whether a certain time period has elapsed. The JVM can query the data while compiling to determine if the current instructions are the most efficient implementation of a bytecode. The JVM can recompile any bytecodes it finds to be executing inefficiently. Once the JVM has determined how a bytecode should be compiled by querying data 607, it can proceed by determining whether it should be the first (default) way or one of the other ways as referred to at step 509 in FIG. 5.

FIG. 7 is a flowchart describing a Java virtual machine compiling a floating point operation and determining how to correct an exception if one arises in accordance with the described embodiment of the present invention. Compiling a floating point operation is a specific example of do

termining how to compile a program in several ways as discussed above. More generally, any application where compilation is guided by heuristics, or assumptions about the average behavior of the code (e.g., compiling TABLESWITCH instructions) can utilize methods of determining how to compile a program in several ways as discussed above. At step 701 the JVM retrieves a bytecode from a Java class file. At step 703 the JVM determines whether the bytecode instruction can create an underflow. Two typical floating point operations that can create an underflow are multiplication and division. In the described embodiment, if the JVM determines that the particular instruction cannot create an underflow problem, it proceeds to compile the bytecode as referred to in step 705.

If the instruction can potentially create an underflow problem, the JVM begins a process of determining how the underflow will be detected and corrected. As discussed above, in the described embodiment, the JVM can use either an explicit check (i.e., inline code) or a trap for detecting underflow. In other embodiments, other methods of detecting underflow can be used in place of or in addition to the described methods.

At step 707 the virtual machine checks whether a counter associated with the trap for detecting underflow has exceeded a predetermined threshold number. As part of the trap, instructions are included to increment a counter associated with each trapping instruction. Instructions are also included to relaunch the bytecode translator if any particular counter exceeds some threshold. The counter is but one example of the type of information or profile data that can be checked to determine which way should be used to compile the particular bytecode, in this example a floating point instruction. Referring to FIG. 6, counters and similar information 607 can be maintained with the native instruction set. In other embodiments, other types of data, such as a timer, can be used in place of or in conjunction with a counter to determine which way the JVM sho

uld use to compile the bytecode.

As referred to, a counter can be updated each time a particular path has been used to execute a specific floating point operation which is being performed multiple times during single execution of a Java class file on a JVM. The counter update can occur, for example, from having the specific instruction in a conditional loop. In the described embodiment, each time a specific floating point operation causes an underflow that is corrected using the trap, a counter is incremented. Referring also to FIG. 5, the "first way", as described at step 509, can correspond to the trap way of compiling an instruction. In the described embodiment, it is desirable to avoid a counter when compiling the first way since it is preferable to reduce the number of operations on the more common path of execution. In the specific embodiment described in FIG. 7, if the counter for the particular floating point operation being compiled (and potentially creating an underflow) has not reached a threshold number, the JVM will continue using the first way. In this example, the trap way of compiling the instruction. If the counter has exceeded a threshold value, the instruction/method is flagged for recompilation a "second way." As described above, as part of the trap, instructions are included to increment a counter associated with each trapping instruction.

The first step in compiling the instruction using the trap way is to determine whether a trap handler has been set up as referred to at step 709. A trap handler is created the first time the trap is called by a Java class file. A trap handler is set up (by the JVM) the first time the compiler decides to compile code that needs the trap handler (the trap handler cannot be needed before that). In the described embodiment, and in most Java programs, there is one trap handler for each thread in the program. For a detailed description of threads, see The Java Language Specification, incorporated herein by reference. If a trap handler is

created. this is done at step 711. If a trap handler is already set up for a particular thread. the JVM compiles the instruction using the trap way as referred to at step 713. This process of compiling the instruction using a trap handler is described in greater detail in FIG. 8. Once compiled, the JVM checks to see if there are any more bytecodes in the Java class files at step 715. If so. the JVM returns to step 701 and repeats the process.

At step 707. the JVM checks if the counter has exceeded a predetermined number (i.e., has the instruction been executed a certain way greater than the predetermined number of times). If so. the JVM compiles the bytecode a next way, which in the example referred to is using an explicit check (inline code) to detect and correct floating point underflow as referred to at step 717. In other embodiments. criteria other than a counter can be used to determine whether the bytecode translator should continue executing a compiled bytecode a certain way. The explicit check way of compiling the bytecode instruction is referred to in greater detail in FIG. 9. At step 718. the virtual machine sets a timer used in conjunction with the explicit check. The time is used to measure the length of time the explicit check way has been used. The virtual machine then checks whether there are any more bytecodes at step 715. If there are none, the process of compiling bytecodes in the Java class files is complete.

FIG. 8 is a flowchart describing in greater detail a process of using a trap to handle underflow from a floating point instruction as referred to in step 713 of FIG. 7. At step 801 the JVM determines which bytecode instruction in the Java class files is invoking the trap. Once the virtual machine determines which instruction is invoking the trap. it increments a counter associated with the floating point operation at step 803. The counter may be maintained with the native instructions as referr

ed to in FIG. 6. Once the counter is incremented, the virtual machine c
hecks the value of the counter at step 805. If the counter is greater t
han a threshold number, the module containing the floating point instru
ction is flagged to be recompiled at step 807.

In the described embodiment, the module is not recompiled immediately.
Instead, it is placed in a queue to be recompiled at a time determined
by the virtual machine based on its resources and level of activity.
In other embodiments, the module can be compiled immediately or at a des
ignated time. Regardless of the actual time the module is recompiled in
the described embodiment, the virtual machine decides to recompile base
d on a counter. In other embodiments, the virtual machine can use other
indicia from the dynamically generated profile data that can be stored
with the native instructions as referred to in FIG. 6. Once the module
has been flagged, or otherwise marked to be recompiled at step 807, the
virtual machine returns to step 715 of FIG. 7 to check whether there are
any more bytecodes in the Java class files. At step 805, if the counte
r has not exceeded a predetermined number, the JVM proceeds by using a t
rap to handle the floating point underflow at step 809. The virtual mac
hine then returns to step 715 of FIG. 7.

FIG. 9 is a flowchart describing an explicit check routine for detect
ing and correcting floating point underflow as referred to in step 717 of
FIG. 7. As described above, an explicit check is code executed in a na
tive instruction set generated from the Java class files by the JVM to d
etect and correct a floating point underflow. FIG. 9 illustrates how a
virtual machine can determine when to compile the explicit check way (wh
ich can correspond to the "next way" as referred to in step 511 of FIG.
5), or when to flag the module containing the floating point instruction
to be recompiled, similarly to step 807 of FIG. 8. At step 901 the vir
tual machine uses the explicit check way to correct the floating point u

nterflow. As described above, explicit counters can be inserted to track how many times underflow was detected by explicit checks, or a timer could be used. One potential drawback to using counters is that they can be relatively expensive in terms of processing. In other embodiments, a combination of a timer and counter can be used. The virtual machine then checks how much time has elapsed from the time the explicit check was first used. Recall that in step 719 of FIG. 7, in the described embodiment, the virtual machine set a timer after the instruction was compiled as an explicit check. The same timer is used in step 902 to determine whether a predetermined time period has elapsed. In the described embodiment, if the predetermined time period has elapsed, the JVM flags or otherwise marks the module to be recompiled at step 905.

In one embodiment, the JVM recompiles the bytecodes after a certain amount of time has passed. This is done in order to reset the currently compiled way back to the default way thereby preventing the accretion of the Java class files from growing potentially inefficient by not adapting to new circumstances. Once the module is flagged and placed in a queue to be recompiled at a time determined by the virtual machine, the counters and other profile data corresponding to the particular flagging put instruction are reset or refreshed so that new profile information can be maintained. The counters are reset at step 907. The virtual machine then returns to step 719 of FIG. 7.

The present invention may employ various computer-implemented operations involving information stored in computer systems. These operations include, but are not limited to, those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities may take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. The operations described herein that form part of the invention are useful machine

operations. The manipulations performed are often referred to in terms such as, producing, identifying, running, determining, comparing, executing, downloading, or detecting. It is sometimes convenient, principally for reasons of common usage, to refer to these electrical or magnetic signals as bits, values, elements, variables, characters, or the like. It should remembered, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities.

The present invention also relates to a device, system or apparatus for performing the aforementioned operations. The system may be specially constructed for the required purposes, or it may be a general purpose computer selectively activated or configured by a computer program stored in the computer. The processes presented above are not inherently related to any particular computer or other computing apparatus. In particular, various general purpose computers may be used with programs written in accordance with the teachings herein, or, alternatively, it may be more convenient to construct a more specialized computer system to perform the required operations.

FIG. 10 is a block diagram of a general purpose computer system 1000 suitable for carrying out the processing in accordance with one embodiment of the present invention. For example, JVM 307, virtual machine 317, or bytecode compiler 309 can run on general purpose computer system 1000. FIG. 10 illustrates one embodiment of a general purpose computer system. Other computer system architectures and configurations can be used for carrying out the processing of the present invention. Computer system 1000, made up of various subsystems described below, includes at least one microprocessor subsystem (also referred to as a central processing unit, or CPU) 1002. That is, CPU 1002 can be implemented by a single-chip processor or by multiple processors. CPU 1002 is a general purpose

digital processor which controls the operation of the computer system 1
00. Using instructions retrieved from memory, the CPU 1002 controls the
reception and manipulation of input information and the output and dis
play of information on output devices.

CPU 1002 is coupled bi-directionally with a first primary storage 1004
, typically a random access memory (RAM), and uni-directionally with a s
econd primary storage area 1006, typically a read only memory (ROM), via
a memory bus 1008. As is well known in the art, primary storage 1004 c
an be used as a general storage area and as scratch-pad memory, and can
also be used to store input data and processed data. It can also store
programming instructions and data, in addition to other data and instruc
tions for processes operating on CPU 1002, and is typically used for fas
t transfer of data and instructions bi-directionally over memory bus 100
8. Also, as is well known in the art, primary storage 1006 typically in
cludes basic operating instructions, program code, data and objects used
by the CPU 1002 to perform its functions. Primary storage devices 1004
and 1006 may include any suitable computer-readable storage media, desc
ribed below, depending on whether, for example, data access needs to be
bi-directional or uni-directional. CPU 1002 can also directly and very
rapidly retrieve and store frequently needed data in a cache memory 1016
.

A removable mass storage device 1012 provides additional data storage
capacity for the computer system 1000, and is coupled either bi-directio
nally or uni-directionally to CPU 1002 via a peripheral bus 1014. For e
xample, a specific removable mass storage device commonly known as a CD
ROM typically passes data uni-directionally to the CPU 1002, whereas a F
loppy disk can pass data bi-directionally to the CPU 1002. Storage 1012
may also include computer-readable media such as magnetic tape, flash m
emory, signals embodied in a carrier wave, Smart Cards, portable mass st

orage devices, and other storage devices. A fixed mass storage 1012 also provides additional data storage capacity and is coupled bi-directionally to CPU 1002 via peripheral bus 1014. Generally, access to these media is slower than access to primary storages 1004 and 1006. Mass storage 1012 and 1016 generally store additional programming instructions, data, and the like that typically are not in active use by the CPU 1002. It will be appreciated that the information retained within mass storages 1012 and 1016 may be incorporated, if needed, in standard fashion as part of primary storage 1004 (e.g. RAM) as virtual memory.

In addition to providing CPU 1002 access to storage subsystems, the peripheral bus 1014 is used to provide access to other subsystems and devices as well. In the described embodiments, these include a display monitor 1018 and adapter 1020, a printer device 1022, a network interface 1024, an auxiliary input/output device interface 1026, a sound card 1028 and speakers 1030, and other subsystems as needed.

The network interface 1024 allows CPU 1002 to be coupled to another computer, computer network, or telecommunications network using a network connection as referred to. Through the network interface 1024, it is contemplated that the CPU 1002 might receive information, e.g., objects, program instructions, or bytecode instructions from a computer in another network, or might output information to a computer in another network in the course of performing the above-described method steps. Information, often represented as a sequence of instructions to be executed on a CPU, may be received from and outputted to another network, for example, in the form of a computer data signal embodied in a carrier wave. An interface card or similar device and appropriate software implemented by CPU 1002 can be used to connect the computer system 1000 to an external network and transfer data according to standard protocols. That is, method embodiments of the present invention may execute solely upon CPU 1002

, or may be performed across a network such as the Internet, intranet networks, or local area networks, in conjunction with a remote CPU that shares a portion of the processing. Additional mass storage devices (not shown) may also be connected to CPU 1002 through network interface 1024.

Auxiliary I/O device interface 1026 represents general and customized interfaces that allow the CPU 1002 to send and, more typically, receive data from other devices. Also coupled to the CPU 1002 is a keyboard controller 1038 via a local bus 1034 for receiving input from a keyboard 1036 or a pointer device 1038, and sending decoded symbols from the keyboard 1036 or pointer device 1038 to the CPU 1002. The pointer device may be a mouse, stylus, track ball, or tablet, and is useful for interacting with a graphical user interface.

In addition, embodiments of the present invention further relate to computer storage products with a computer readable medium that contain program code for performing various computer-implemented operations. The computer-readable medium is any data storage device that can store data which can thereafter be read by a computer system. Examples of computer-readable media include, but are not limited to, all the media mentioned above, including hard disks, floppy disks, and specially configured hardware devices such as application-specific integrated circuits (ASICs), or programmable logic devices (PLDs). The computer-readable medium can also be distributed as a data signal embedded in a carrier wave over a network of coupled computer systems so that the computer-readable code is stored and executed in a distributed fashion.

It will be appreciated by those skilled in the art that the above described hardware and software elements are of standard design and construction. Other computer systems suitable for use with the invention may include additional or fewer subsystems. In addition, memory bus 1006, peripheral bus 1014, and local bus 1034 are illustrative of any interconnect

ion scheme serving to bridge the subsystems. For example, a north bridge would be used to connect the CPU to fixed mass storage 1036 and display adapter 1020. The computer system referred to in FIG. 10 is but an example of a computer system suitable for use with the invention. Other computer architectures having different configurations of subsystems may also be utilized.

Although the foregoing invention has been described in some detail for purposes of clarity of understanding, it will be apparent that certain changes and modifications may be practiced within the scope of the appended claims. For example, although the trap and explicit inline check ways are described with regard to floating point underflow, other techniques for detecting underflow can also be used and incorporated into the present invention. In another example, although two ways for compiling an instruction are described, the methods and apparatus of the present invention can accommodate more than two ways for compiling a program if more ways are available. Moreover, it should be noted that although the present invention has been illustrated using floating point underflow operations, the present invention can choose intelligently and dynamically an implementation to use in solving a problem arising from platform-specific variations. Floating point underflow is only one such problem. Furthermore, it should be noted that there are alternative ways of implementing both the process and apparatus of the present invention. Accordingly, the present embodiments are to be considered as illustrative and not restrictive, and the invention is not to be limited to the details given herein, but may be modified within the scope and equivalents of the appended claims.

4 Brief Description of Drawings

FIG. 1 is a block diagram showing a typical format of double precision

floating point and the format of exercise precision floating point as known in the prior art.

FIGS. 2a and 2b illustrate two methods of detecting floating point underflow as are known in the prior art.

FIG. 3a is a block/process diagram illustrating the transformation of a Java (trademark) program containing Java source code to native code to be run on a particular platform or computer.

FIG. 3b is a diagrammatic representation of virtual machine 201, supported by computer system 1000 of FIG. 10 described below.

FIG. 4 is a block diagram showing how different versions of native instructions can be generated from a Java program.

FIG. 5 is a flowchart showing a process of a Java virtual machine compiling Java bytecodes in accordance with one embodiment of the present invention.

FIG. 6 is a block diagram illustrating how native instructions including dynamically generated profile data are generated in accordance with one embodiment of the present invention.

FIG. 7 is a flowchart showing a Java virtual machine compiling a floating point operation and determining how to correct an underflow if one arises in accordance with one embodiment of the present invention.

FIG. 8 is a flowchart showing a process of using a trap routine to handle underflow from a floating point instruction as referred to in step 713 of FIG. 7 in greater detail.

FIG. 9 is a flowchart showing an explicit check routine for detecting and correcting floating point underflows as referred to in step 713 of FIG. 7.

FIG. 10 is a block diagram of a typical computer system suitable for implementing an embodiment of the present invention.

102
104
+
-
106
EXPONENT: 15 bits
108
SIGNIFICAND: 64 bits
112
104
+
-
114
EXPONENT: 11 bits
116
SIGNIFICAND: 52 bits
118

FIG. 1

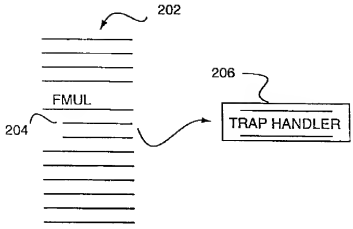


FIG. 2a

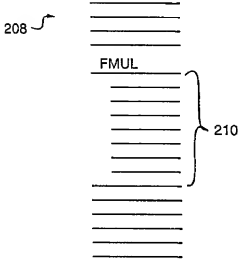


FIG. 2b

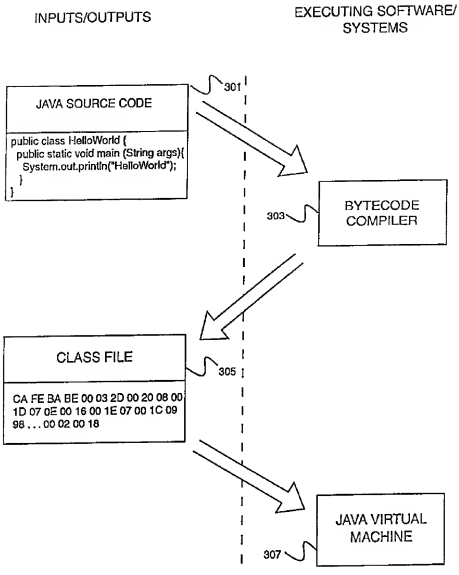

INPUTS/OUTPUTS
EXECUTING SOFTWARE/
SYSTEMS
JAVA SOURCE CODE
public class HelloWorld {
public static void main (String args){
System.out.println("HelloWorld");
}
}
301
BYTECODE
COMPILER
303
CLASS FILE
CA FE BA BE 00 03 2D 00 20 08 00
1D 07 0E 00 16 00 1E 07 00 1C 09
96 . . . 00 02 00 18
305
JAVA VIRTUAL
MACHINE
307

FIG. 3a

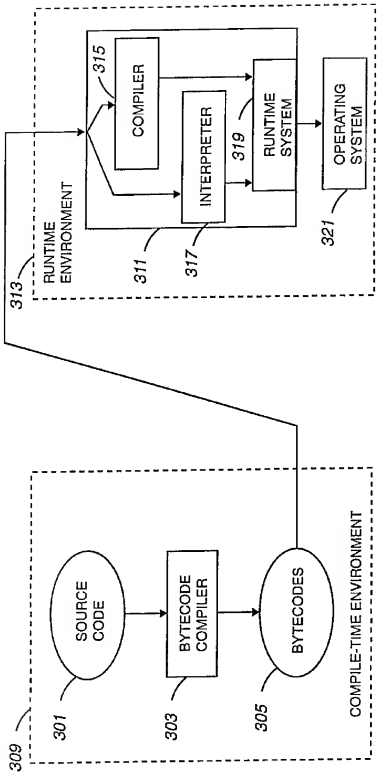

309
301
SOURCE CODE
303
BYTECODE COMPILER
305
BYTECODES
COMPILE-TIME ENVIRONMENT
313
RUNTIME ENVIRONMENT
311
315
COMPILER
317
INTERPRETER
319
RUNTIME SYSTEM
321
OPERATING SYSTEM

FIG. 3b

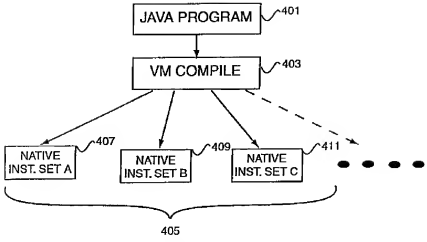

JAVA PROGRAM
401
VM COMPILE
403
NATIVE INST. SET A
407
NATIVE INST. SET B
409
NATIVE INST. SET C
411
405

FIG. 4

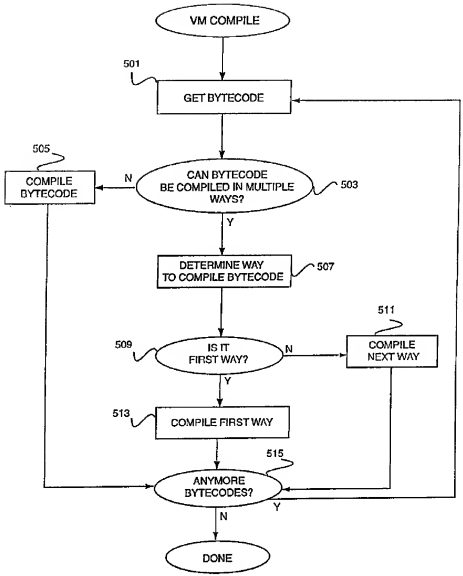

VM COMPILE
501
GET BYTECODE
505
COMPILE BYTECODE
N
CAN BYTECODE BE COMPILED IN MULTIPLE WAYS?
503
Y
DETERMINE WAY TO COMPILE BYTECODE
507
511
509
IS IT FIRST WAY?
N
COMPILE NEXT WAY
Y
513
COMPILE FIRST WAY
515
ANYMORE BYTECODES?
Y
N
DONE

FIG. 5

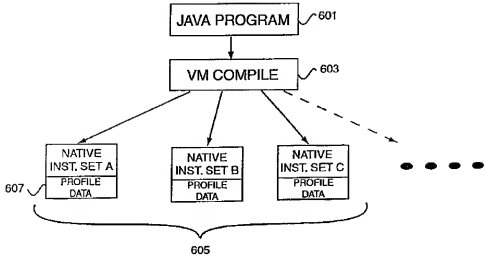

JAVA PROGRAM
601
VM COMPILE
603
NATIVE INST. SET A
PROFILE DATA
NATIVE INST. SET B
PROFILE DATA
NATIVE INST. SET C
PROFILE DATA
607
605

FIG. 6

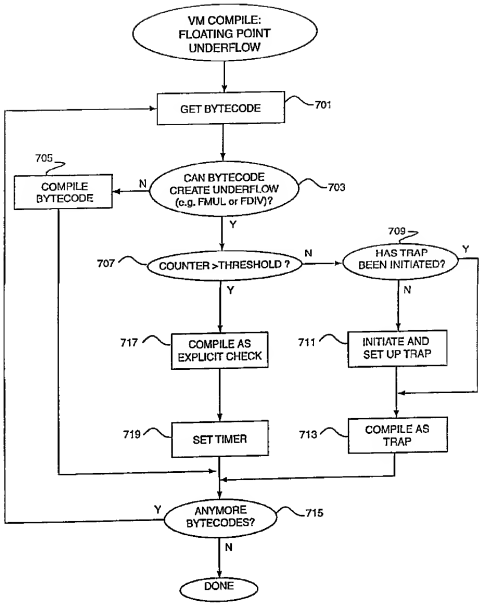

VM COMPILE: FLOATING POINT UNDERFLOW
GET BYTECODE
701
CAN BYTECODE CREATE UNDERFLOW (e.g. FMUL or FDIV)?
703
N
Y
COMPILE BYTECODE
705
COUNTER >THRESHOLD ?
707
N
Y
HAS TRAP BEEN INITIATED?
709
Y
N
COMPILE AS EXPLICIT CHECK
717
INITIATE AND SET UP TRAP
711
SET TIMER
719
COMPILE AS TRAP
713
ANYMORE BYTECODES?
715
Y
N
DONE

FIG. 7

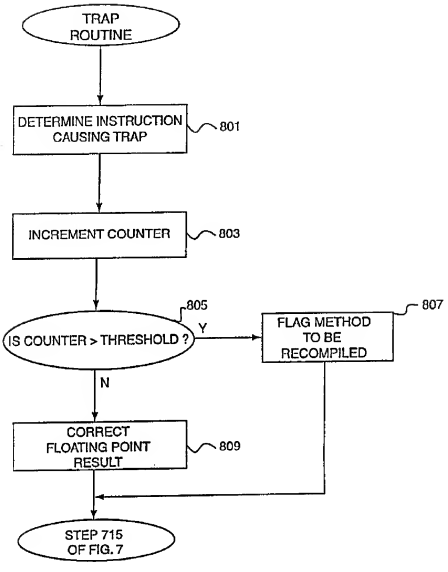

TRAP ROUTINE
DETERMINE INSTRUCTION CAUSING TRAP
801
INCREMENT COUNTER
803
805
IS COUNTER > THRESHOLD ?
Y
N
FLAG METHOD TO BE RECOMPILED
807
CORRECT FLOATING POINT RESULT
809
STEP 715 OF FIG. 7

FIG. 8

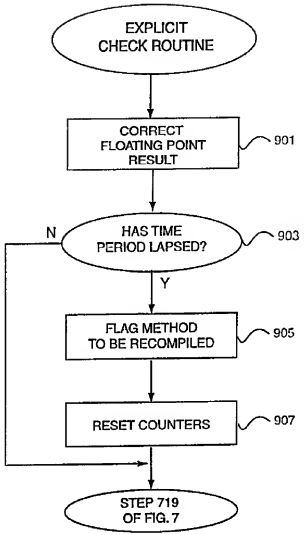

EXPLICIT CHECK ROUTINE
CORRECT FLOATING POINT RESULT
901
HAS TIME PERIOD LAPSED?
903
N
Y
FLAG METHOD TO BE RECOMPILED
905
RESET COUNTERS
907
STEP 719 OF FIG. 7

FIG. 9

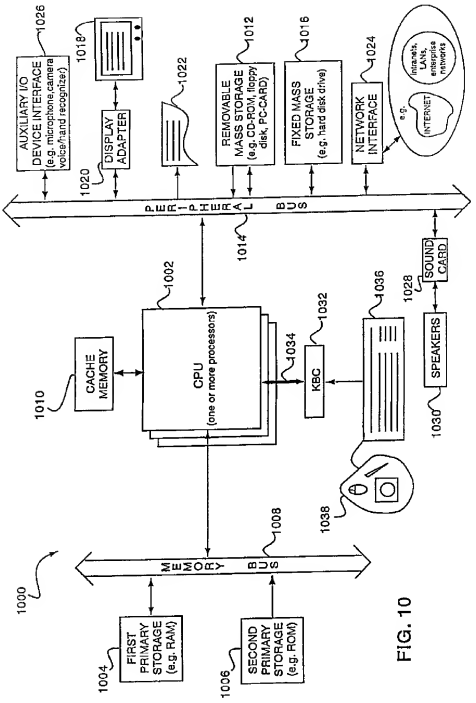

1000
MEMORY BUS
1008
1004
FIRST PRIMARY STORAGE (e.g. RAM)
1006
SECOND PRIMARY STORAGE (e.g. ROM)
1010
CACHE MEMORY
CPU (one or more processors)
1002
1034
KBC
1032
1036
1038
1028
SOUND CARD
1030
SPEAKERS
PERIPHERAL BUS
1014
1012
REMOVABLE MASS STORAGE (e.g. CD-ROM, floppy disk, PC-CARD)
1016
FIXED MASS STORAGE (e.g. hard disk drive)
1024
NETWORK INTERFACE
e.g.
(INTERNET)
intranets, LANs, enterprise networks
1022
1020
DISPLAY ADAPTER
1018
AUXILIARY I/O DEVICE INTERFACE (e.g. microphone,camera voice/hand recognizer)
1026

FIG. 10

1 Abstract:

Apparatus, methods, and computer program products are disclosed for determining how to compile a program at runtime. A bytecode instruction or associated with the program that can be compiled in multiple ways is retrieved and compiled in a particular way, typically the default way. At runtime, a virtual machine determines whether another way of compiling the bytecode instruction is more desirable and, if so, the bytecode is then recompiled the other way. In some embodiments, the portion of the program that contains the bytecode instruction to be recompiled is placed in a queue with other instructions that are to be recompiled. The virtual machine may examine changing requirements of the program that have developed at the program's execution in which the requirements are derived from profile data on each of the multiple ways the program can be compiled. The bytecode instruction within the program may be recompiled in a more preferred way based upon the profile data.

2 Representative Drawing

Fig. 5